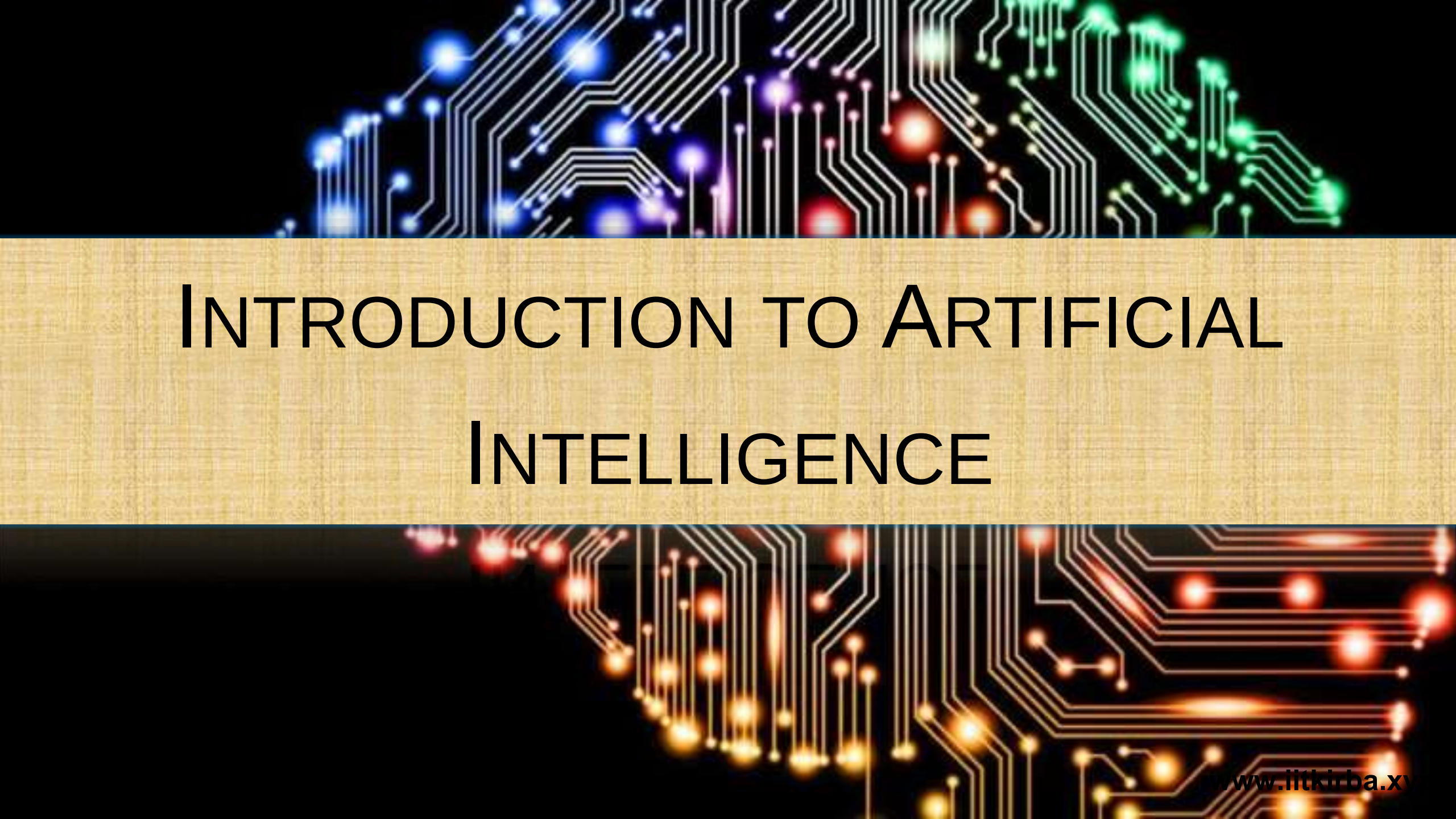
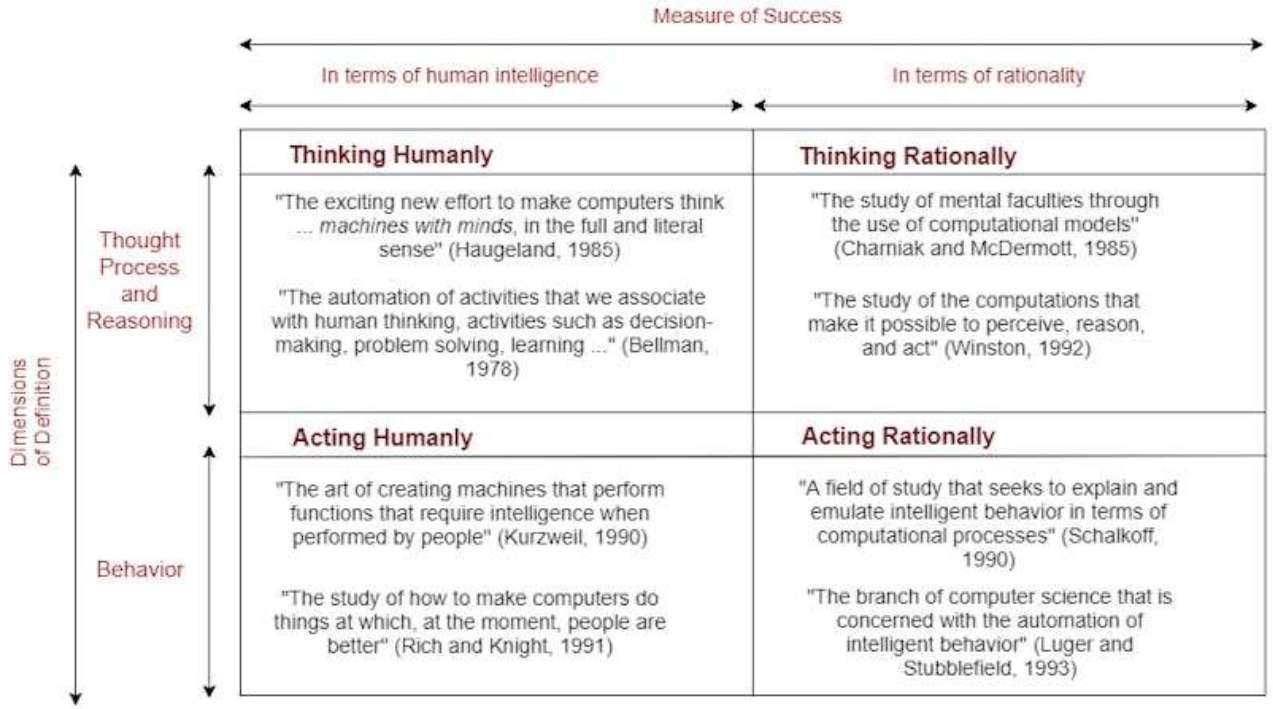


Introduction to Artificial Intelligence

ARTIFICIAL INTELLIGENCE

Artificial intelligence, as a computer science discipline, works to develop machines that execute duties that require human cognitive abilities. The human-related operations encompass learning combined with reasoning alongside problem-solving and perception and decision-making paths.

“It is a branch of computer science by which we can create intelligent machines that can behave like a human, think like humans, and be able to make decisions.”



ARTIFICIAL INTELLIGENCE

- **Systems that can think humanly**
  - For following this approach, ***one first needs to understand how humans think.*** Knowing the internal working of human brain can be achieved by introspection or psychological experiments. This, in itself, is a vast interdisciplinary field, ***known as Cognitive Science.***
- **Systems that can act humanly**
  - *This definition came into being when* ***Alan Turing proposed the Turing Test***. A system passes this test, if it can fool a human interrogator by depicting intelligent behavior. *By intelligent behavior, we mean achieving human level performance in cognitive tasks.* Such a system would require to have major components of A.I., including natural language processing, knowledge representation, automated reasoning, machine learning, robotics and computer vision. Seeing the underlying complications, no major effort has been made in trying to make such a machine.

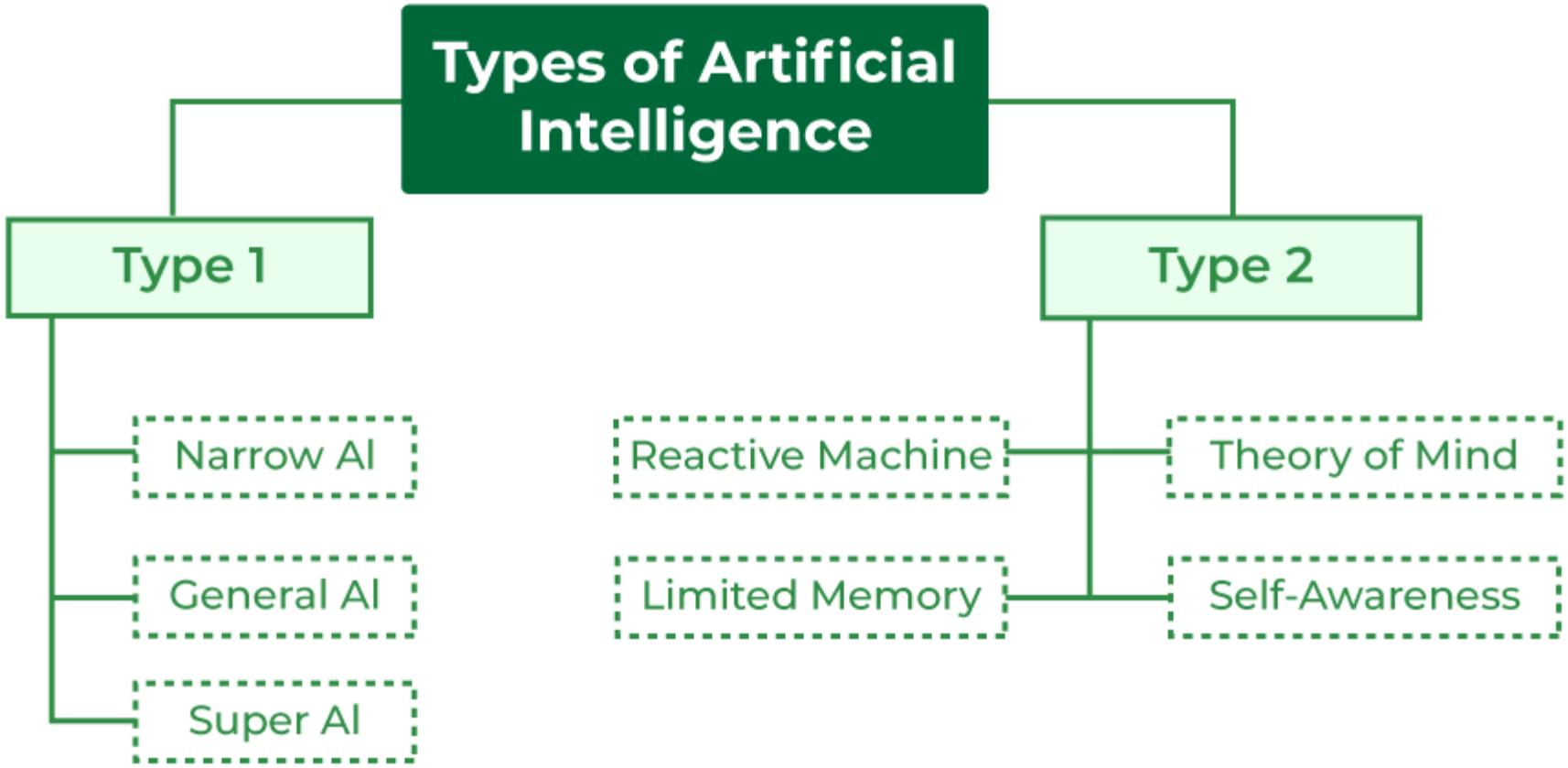
ARTIFICIAL INTELLIGENCE

- **Systems that can think rationally**
  - ***The aim of this approach is to build upon programs that represent "right thinking", to create intelligent systems.*** This "right thinking" or irrefutable reasoning processes, is defined in coding (in mathematical terms) using **logic or laws of thought**.

1. ***Not all knowledge can be expressed with logical notations (especially when knowledge is not 100% certain).***
2. ***It can lead to computational blow up, as without guidance, there are many reasoning steps that can be tried.***

- **Systems that can act rationally**
  - *This approach involves creating systems that act in a way which maximizes its chances of achieving its goal, given the available information.* These systems are known as **Rational Agents,** such that ***they perceive the***

Types of Artificial Intelligence



www.iitkirba.xyz

AI Type 1: Based on Capabilities

1.Weak AI or Narrow AI: This type of AI can be used to solve certain problems and focus on a particular kind of job. It is only effective when it is used in a specific area and does not produce the same results when applied in other areas. It applies to smart products, known as virtual assistants such as Siri, systems engaged in image recognition, and IBM's Watson.

2.General AI: General AI, also known as Strong AI, on the other hand, refers to machines capable of achieving any action that a man is capable of accomplishing. It is planning to attain human-like features such as intelligence characteristics like reasoning and learning processes. This is another type of AI that is still under research and has not been developed to its realization.

3.Super AI: An advanced artificial intelligence in which every domain is superior to that of humans in terms of their decision-making power, problem-solving skills, learning capabilities, as well as their feelings and emotions. It is the final stage of AI development, and it does not currently exist in the world.

AI Type 2: Based on Functionality

1.Reactive Machines: This type of AI processes the current input data and does not have any previous experience. They follow pre-defined rules. Some of the most widely known examples include IBM's chess machine known as Deep Blue and the Go-playing computer termed Google's AlphaGo.

2.Limited Memory: Many of them use the earlier information to establish something for a limited period. Some of the concrete samples include self-driving cars that follow other vehicles, the speed, and the road condition of the environment.

3.Theory of Mind: The purpose of this AI is to comprehend the feelings, desires or even gestures of people. As previously stated, it is still part of the theoretical research and has not been fully realized.

4.Self-Awareness: The final type of artificial intelligence that remains at the level of theory is even more superior to human intelligence as it would have consciousness and feelings. People would consider this level of AI as a significant level of advancement in technology as well as in knowledge.

PROBLEMS OF AI

Intelligence does not imply perfect understanding; every intelligent being has limited perception, memory and computation. Many points on the spectrum of intelligence versus cost are viable, from insects to humans. AI seeks to understand the computations required from intelligent behaviour and to produce computer systems that exhibit intelligence. Aspects of intelligence studied by AI include perception, communicational using human languages, reasoning, planning, learning and memory. The following questions are to be considered before we can step forward:

1. What are the underlying assumptions about intelligence?
2. What kinds of techniques will be useful for solving AI problems?
3. At what level human intelligence can be modelled?
4. When will it be realized when an intelligent program has been built?

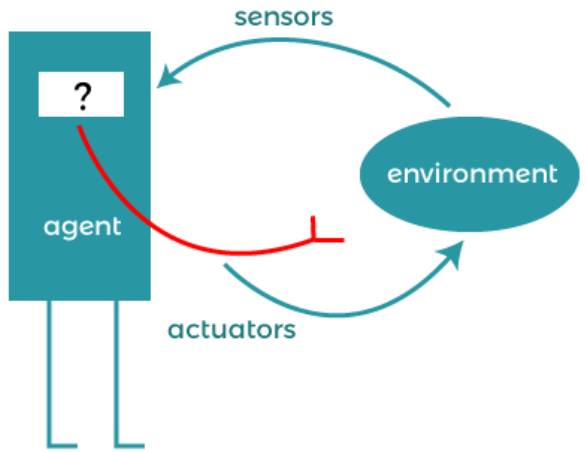
APPLICATIONS OF AI

AI has applications in all fields of human study, such as finance and economics, environmental engineering, chemistry, computer science, and so on. Some of the applications of AI are listed below:

- Perception
  - Machine vision
  - Speech understanding
  - Touch (tactile or haptic) sensation
- Robotics
- Natural Language Processing
  - Natural Language Understanding
  - Speech Understanding
  - Language Generation
  - Machine Translation
- Planning
- Expert Systems
- Machine Learning
- Theorem Proving
- Symbolic Mathematics
- Game Playing

AI AGENTS

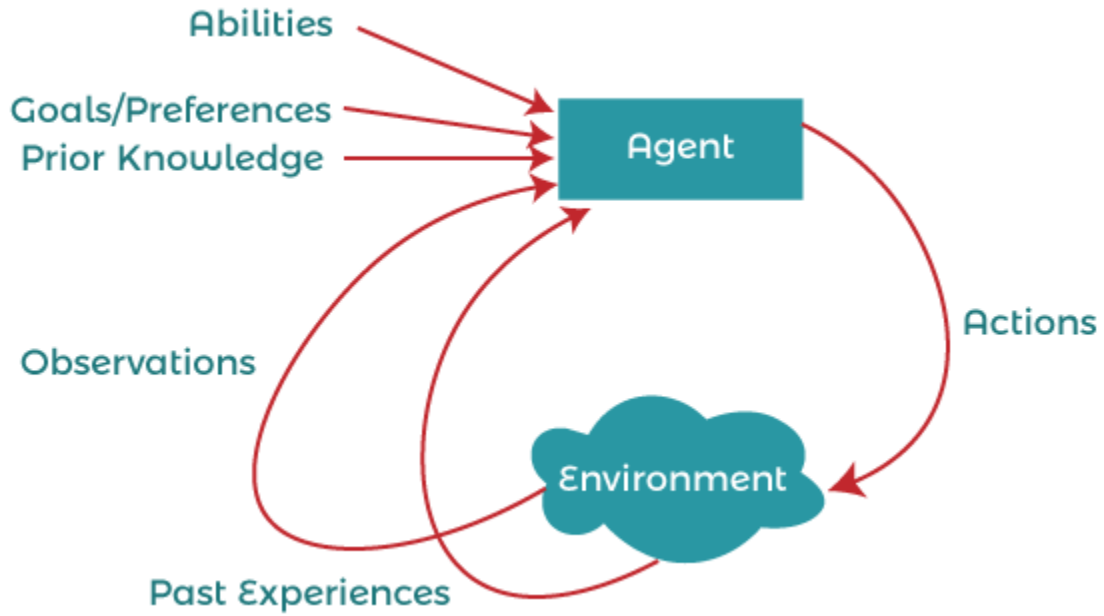
A rational agent may take the form of a person, firm, machine, or software to make decisions. It works with the best results after considering past and present perceptions (perceptual inputs of the agent at a given instance). An AI system is made up of an agent and its environment. Agents work in their environment, and the environment may include other agents.



agent = architecture + agent program

- A software agent has keystrokes, file contents, received network packages that act as sensors and are displayed on the screen, files, sent network packets to act as actuators.
- The human agent has eyes, ears, and other organs that act as sensors, and hands, feet, mouth, and other body parts act as actuators.
- A robotic agent consists of cameras and infrared range finders that act as sensors and various motors that act as actuators.

AI AGENTS



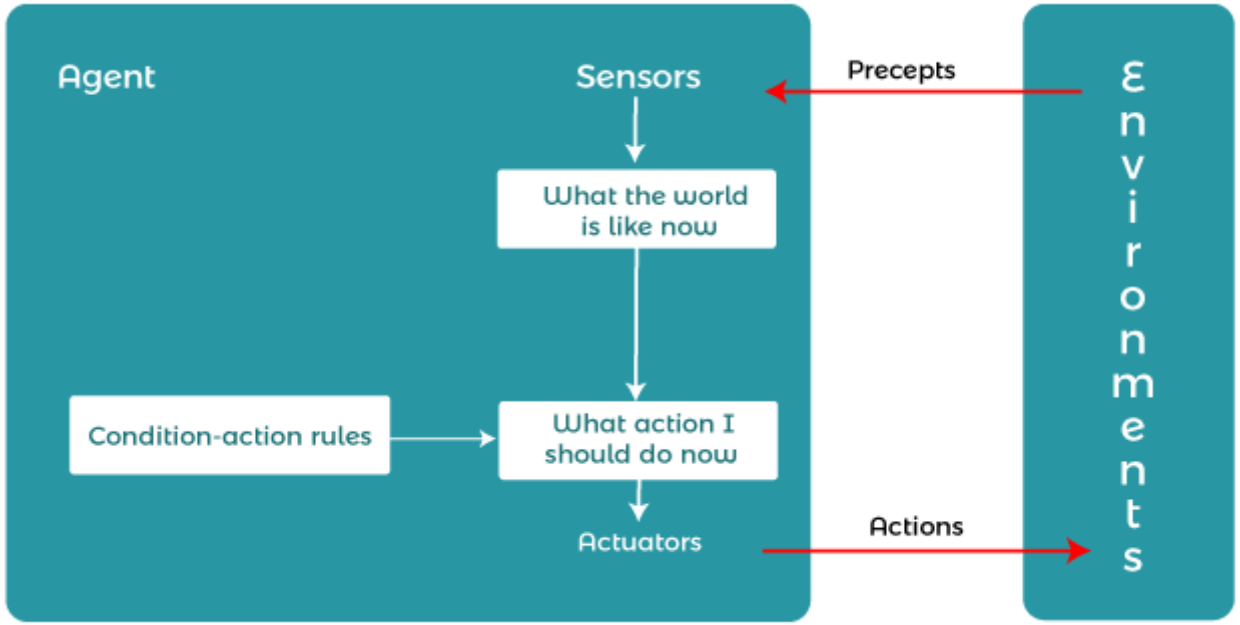
Types of agents

Agents can be divided into four classes based on their perceived intelligence and ability:

- Simple reflex agent
- Model-based reflex agent
- Target-based agent
- Utility-based agent
- Learning agent

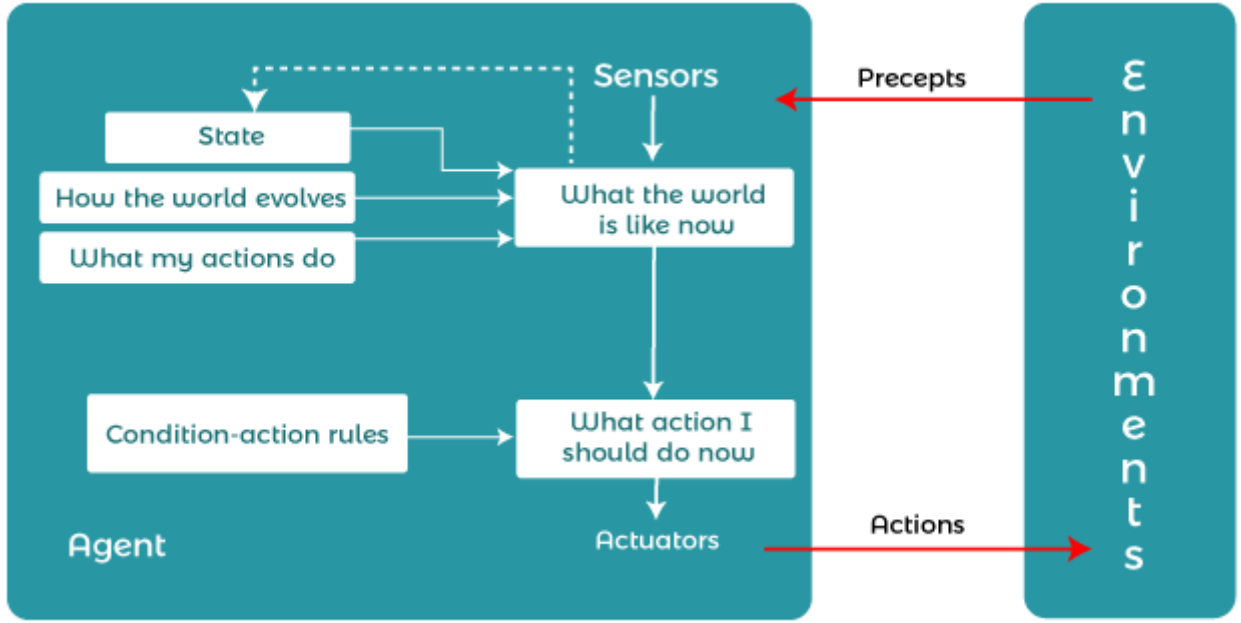
AI AGENTS

Simple reflex agent: Simple reflex agents ignore the rest of the concept history and act only based on the current concept. Concept history is the history of all that an agent has believed to date. The agent function is based on the condition-action rule. A condition-action rule is a rule that maps a state, that is, a condition, to an action. If the condition is true, then action is taken; otherwise, not.



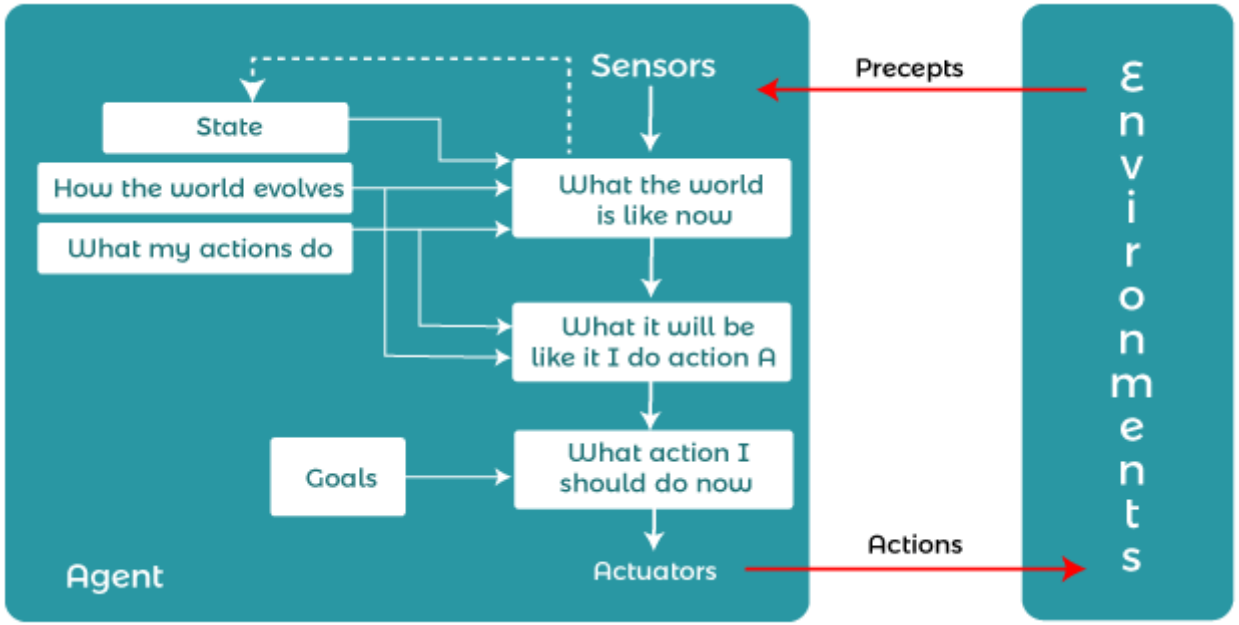
AI AGENTS

Model-based reflex agents: It works by searching for a rule whose position matches the current state. A model-based agent can handle a partially observable environment using a model about the world. The agent has to keep track of the internal state, adjusted by each concept, depending on the concept history. The current state is stored inside the agent, which maintains some structure describing the part of the world that cannot be seen.



AI AGENTS

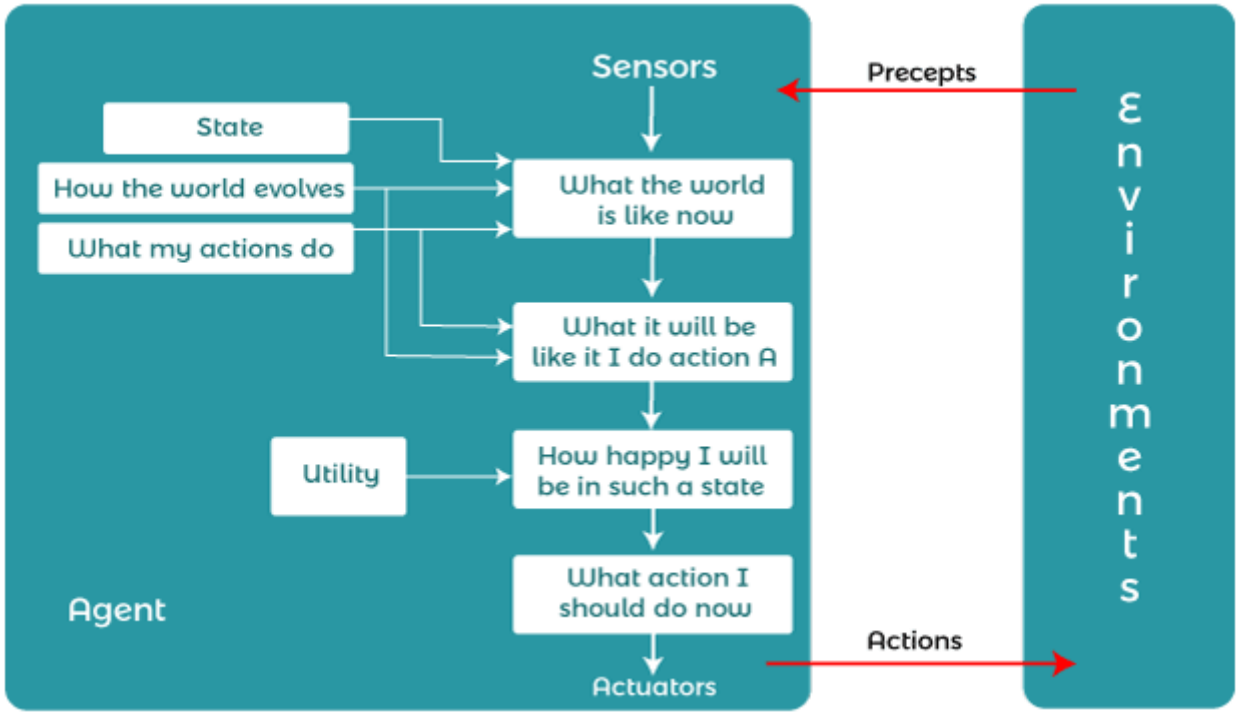
Goal-based agents: These types of agents make decisions based on how far they are currently from their goals (details of desired conditions). Their every action is aimed at reducing its distance from the target. This gives the agent a way to choose from a number of possibilities, leading to a target position. The knowledge supporting their decisions is clearly presented and can be modified, which makes these agents more flexible.



AI AGENTS

Utility-based agents: The agents which are developed having their end uses as building blocks are called utility-based agents. When there are multiple possible alternatives, then to decide which one is best, utility-based agents are used. They choose actions based on a **preference (utility)** for each state. Sometimes achieving the desired goal is not enough. We may look for a quicker, safer, cheaper trip to reach a destination. Agent happiness should be taken into consideration.

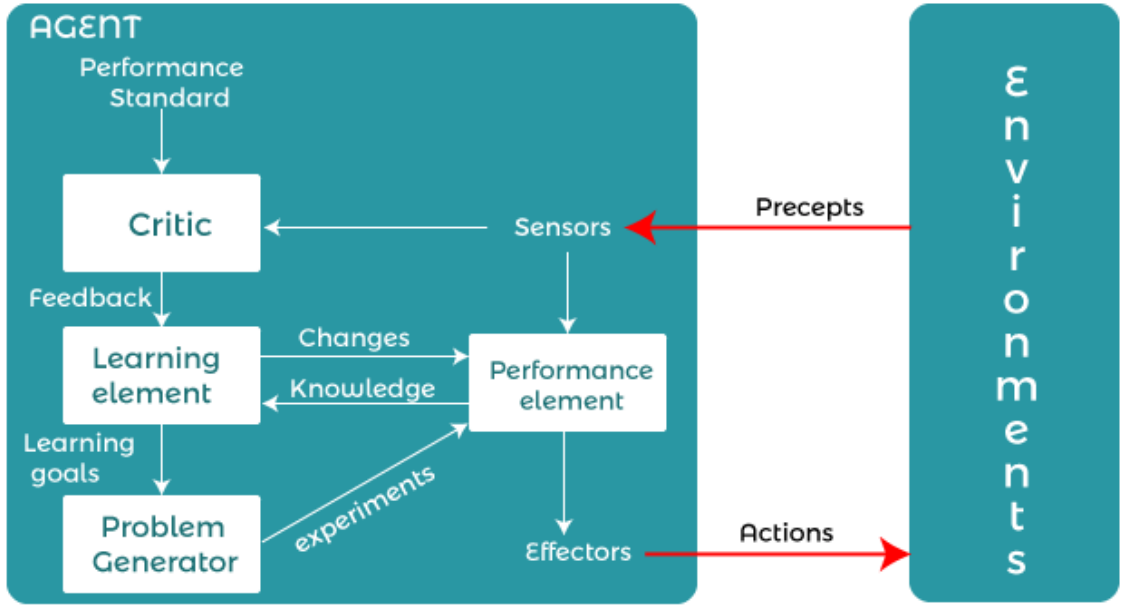
Utility describes how **"happy"** the agent is. Because of the uncertainty in the world, a utility agent chooses the action that maximizes the expected utility. A utility function maps a state onto a real number which describes the associated degree of happiness.



AI AGENTS

Learning Agent: A learning agent in AI is the type of agent that can learn from its past experiences or it has learning capabilities. It starts to act with basic knowledge and then is able to act and adapt automatically through learning.
A learning agent has mainly four conceptual components, which are:

- **Learning element:** It is responsible for making improvements by learning from the environment
- **Critic:** The learning element takes feedback from critics which describe how well the agent is doing with respect to a fixed performance standard.
- **Performance element:** It is responsible for selecting external action
- **Problem Generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.



PEAS IN ARTIFICIAL INTELLIGENCE

PEAS stands for performance measure, environment, actuators, and sensors. PEAS defines AI models and helps determine the task environment for an intelligent agent.

- **Performance measure:** It defines the success of an agent. It evaluates the criteria that determines whether the system performs well.
- **Environment:** It refers to the external context in which an AI system operates. It encapsulates the physical and virtual surroundings, including other agents, objects, and conditions.
- **Actuators:** They are responsible for executing actions based on the decisions made. They interact with the environment to bring about desired changes.
- **Sensors:** An agent observes and perceives its environment through sensors. Sensors provide input data to the system, enabling it to make informed decisions.

PEAS in Artificial Intelligence

Agent	Performance measure	Environment	Actuators	Sensors
Vacuum cleaner	Cleanliness, security, battery	Room, table, carpet, floors	Wheels, brushes	Camera, sensors
Chatbot system	Helpful responses, accurate responses	Messaging platform, internet, website	Sender mechanism, typer	NLP algorithms
Autonomous vehicle	Efficient navigation, safety, time, comfort	Roads, traffic, pedestrians, road signs	Brake, accelerator, steer, horn	Cameras, GPS, speedometer
Hospital	Patient's health, cost	Doctors, patients, nurses, staff	Prescription, diagnosis, tests, treatments	Symptoms

PROBLEM, PROBLEM SPACE, SEARCH

A **problem** is a specific task or challenge that requires finding a solution or making a decision. In artificial intelligence, problems can vary in complexity and scope, ranging from simple tasks like arithmetic calculations to complex challenges such as image recognition, natural language processing, game playing, and optimization. Each problem has a defined set of initial states, possible actions or moves, and a goal state that needs to be reached or achieved.

The **problem space** is the set of all possible states, actions, and transitions that can be encountered while attempting to solve a specific problem. It represents the entire landscape of potential solutions and paths from the initial state to the goal state. In other words, the problem space defines all the possible configurations or arrangements of elements involved in the problem and the set of valid moves or actions that can be taken at each state. Each state in the problem space represents a specific configuration, and each action represents a possible move or step from one state to another.

Search is the process of exploring the problem space to find a sequence of actions or moves that lead to the goal state or a satisfactory solution. In AI, search algorithms are used to systematically navigate through the problem space and discover paths or solutions that satisfy the problem's constraints and objectives.

PROBLEM CHARACTERISTICS

- **Is the problem decomposable?**

The question of whether a problem is decomposable refers to whether it can be broken down into smaller, independent subproblems. Decomposable problems can be solved by tackling each subproblem individually, and their solutions can then be combined to solve the overall problem. Some problems, like complex integrals, can be decomposed into simpler subproblems, making it easier to find a solution using the divide-and-conquer approach. However, not all problems are decomposable, and some may require addressing as a whole without breaking them down into independent parts.

- **Can solution steps be ignored or undone?**

The reversibility of solution steps refers to whether they can be ignored or undone if they prove to be unwise or lead to a dead end. In some problems, certain solution steps can be ignored without affecting the final result. In recoverable problems, solution steps can be undone to explore alternative paths. For instance, in the 8-puzzle, moves can be undone to try different arrangements of tiles. On the other hand, some problems have irreversible solution steps, like in chess, where once a move is made, it cannot be undone.

PROBLEM, PROBLEM SPACE, SEARCH

- **Is the problem's universe predictable?**

The predictability of a problem's universe refers to whether the outcomes or states of the problem can be determined with certainty or if they involve uncertainty. Some problems have deterministic outcomes, meaning that the result is known and can be predicted with complete certainty based on the given conditions and rules. Other problems may involve uncertainty or randomness, leading to non-deterministic outcomes. For example, some optimization problems may have multiple potential solutions with different probabilities of being optimal.

- **Is a good solution absolute or relative?**

The nature of a good solution can be either absolute or relative. An absolute solution is one where finding a single correct path or outcome is sufficient to achieve the desired goal. In problems like the water jug puzzle, finding any valid path to the solution is considered good enough. On the other hand, a relative solution is one that requires evaluating multiple possible paths or outcomes to find the best or optimal solution. Problems like the traveling salesman problem seek the shortest route among all possible routes, making it a relative solution.

PROBLEM, PROBLEM SPACE, SEARCH

- **Is the solution a state or a path?**

The solution to a problem can be either a state or a path, depending on the nature of the problem. In some problems, the desired outcome is a specific state or configuration that satisfies the problem's requirements. For instance, in the 8-puzzle, the solution is a specific arrangement of tiles in the goal state. In other problems, the solution involves finding a path or sequence of steps to reach the desired goal state. For example, in maze solving, the solution is the path from the starting point to the exit.

- **What is the role of knowledge?**

The role of knowledge in problem-solving varies based on the complexity and nature of the problem. Knowledge plays a critical role in guiding the problem-solving process. In some problems, extensive domain specific knowledge is required to recognize patterns, constraints, and possible solutions. For example, chess requires deep knowledge of the game rules and strategic principles to make informed moves. In contrast, other problems may rely more on general problem-solving algorithms and heuristics, requiring less domain-specific knowledge.

- **Can a computer give the problem solution, or interaction with humans is required?**

The level of human interaction required in problem-solving depends on the problem's complexity and the capabilities of the problem-solving methods being used. In some cases, computers can autonomously find solutions to problems without any interaction with humans. For example, algorithms can efficiently solve mathematical equations or perform certain optimization tasks. However, in more complex and uncertain problems, human interaction may be necessary to provide additional information, preferences, or guidance. Conversational problem-solving, where the computer interacts with users to gather information or provide assistance, can be valuable in addressing such challenges.

STATE SPACE

A **state space** is a way to mathematically represent a problem by defining all the possible states in which the problem can be. This is used in search algorithms to represent the initial state, goal state, and current state of the problem. Each state in the state space is represented using a set of variables.

State space search has several features that make it an effective problem-solving technique in Artificial Intelligence.

These features include:

- **Exhaustiveness:** State space search explores all possible states of a problem to find a solution.
- **Completeness:** If a solution exists, state space search will find it.
- **Optimality:** Searching through a state space results in an optimal solution.
- **Uninformed and Informed Search:** State space search in artificial intelligence can be classified as uninformed if it provides additional information about the problem.

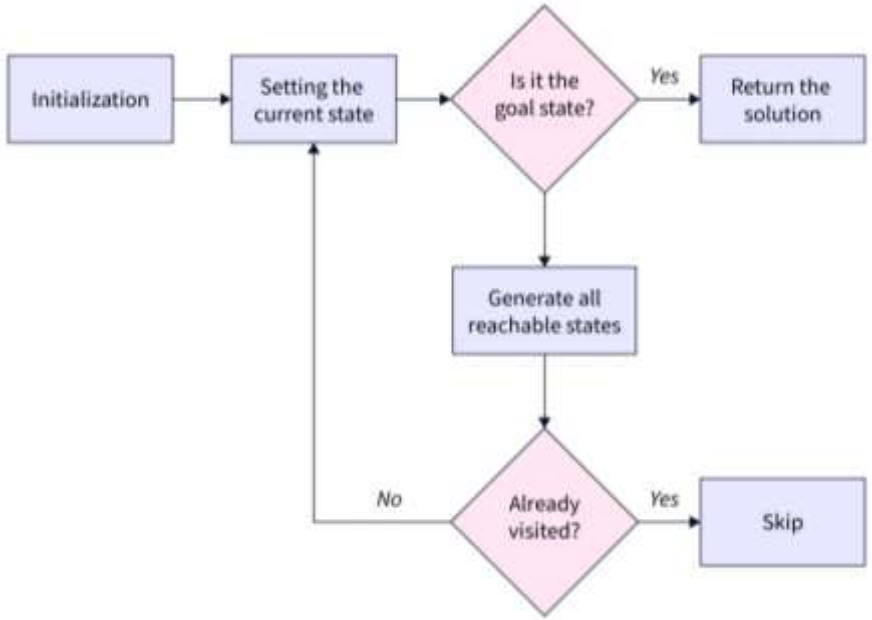
STATE SPACE REPRESENTATION

State space Representation involves defining an INITIAL STATE and a GOAL STATE and then determining a sequence of actions, called states, to follow.

- **State:** A state can be an Initial State, a Goal State, or any other possible state that can be generated by applying rules between them.
  - **Initial State:** The state of the issue as it first arises.
  - **Goal State:** The idealized final state that delineates a problem-solving strategy.
  - **Operators:** The collection of maneuvers or actions that can be used to change a state.
  - **Restrictions:** Guidelines or limitations must be adhered to to solve the problem.
- **Space:** In an AI problem, space refers to the exhaustive collection of all conceivable states.
- **Search:** This technique moves from the beginning state to the desired state by applying good rules while traversing the space of all possible states.
- **Search Tree:** To visualize the search issue, a search tree is used, which is a tree-like structure that represents the problem. The initial state is represented by the root node of the search tree, which is the starting point of the tree.
- **Transition Model:** This describes what each action does, while Path Cost assigns a cost value to each path, an activity sequence that connects the beginning node to the end node. The optimal option has the lowest cost among all alternatives.

State Space Representation

- To begin the search process, we set the current state to the initial state.
- We then check if the current state is the goal state. If it is, we terminate the algorithm and return the result.
- If the current state is not the goal state, we generate the set of possible successor states that can be reached from the current state.
- For each successor state, we check if it has already been visited. If it has, we skip it, else we add it to the queue of states to be visited.
- Next, we set the next state in the queue as the current state and check if it's the goal state. If it is, we return the result. If not, we repeat the previous step until we find the goal state or explore all the states.
- If all possible states have been explored and the goal state still needs to be found, we return with no solution.



Example of State Space

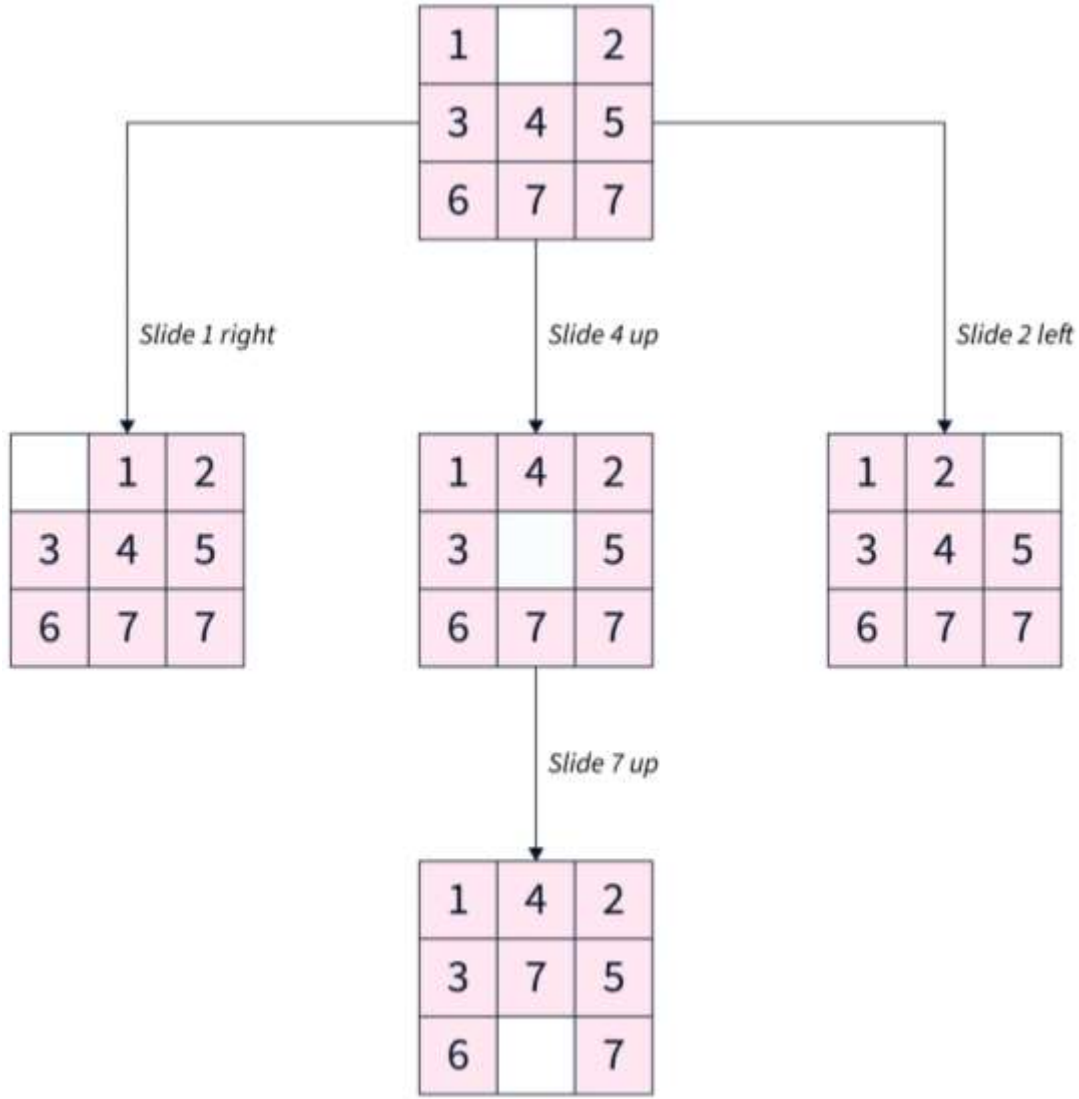
8-puzzle problem

1		2
3	4	5
6	7	7

Current State

1	4	2
3	7	5
6		7

Target State



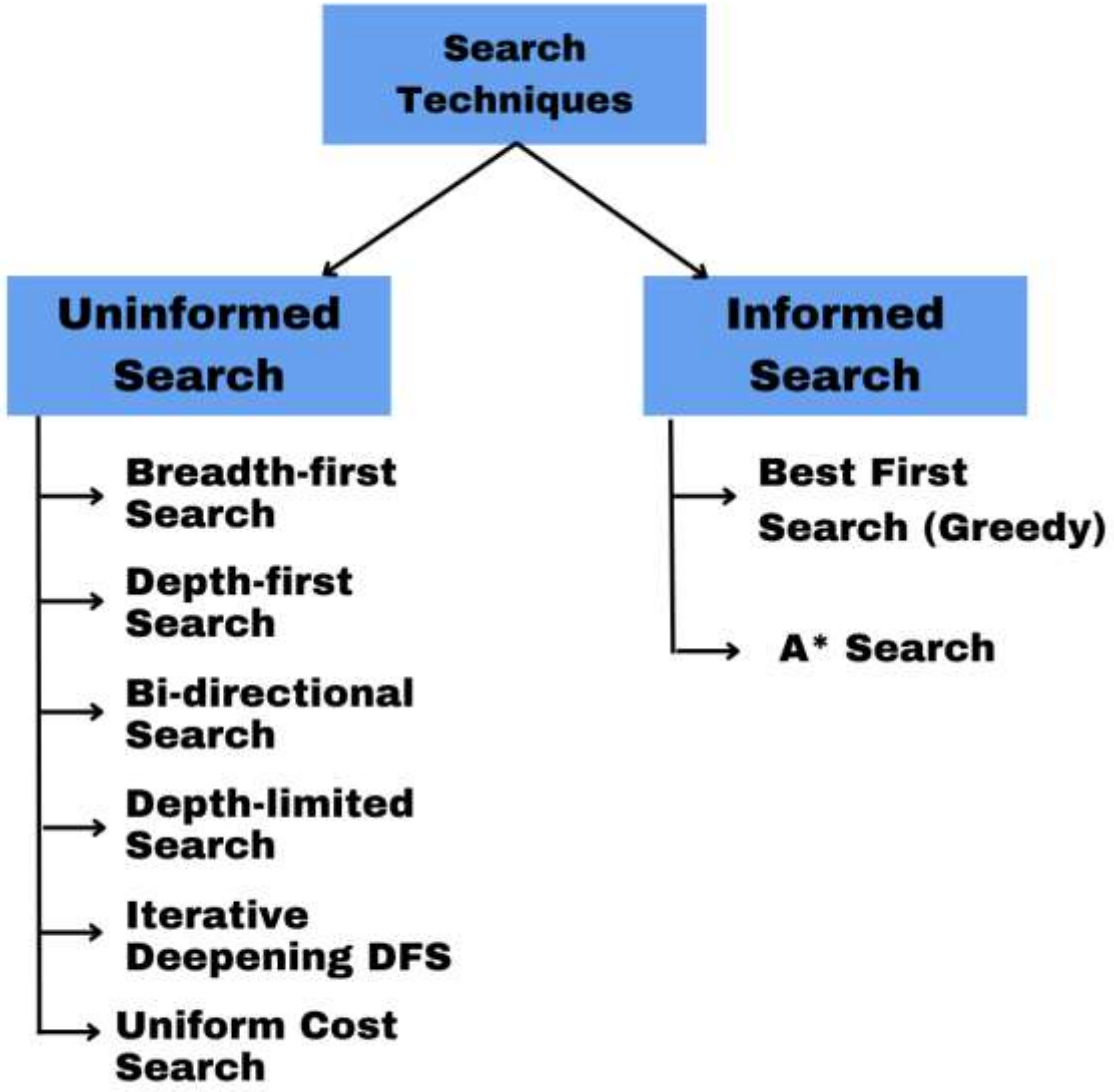
STATE SPACE SEARCH

Search algorithms in AI provide search solutions by transforming the initial state to the desired state.

Properties of Search Algorithms: The four important properties of search algorithms in artificial intelligence for comparing their efficiency are as follows:

- **Completeness** - A search algorithm is said to be complete if it guarantees to yield a solution for any random input if at least one solution exists.
- **Optimality** - A solution discovered for an algorithm is considered optimal if it is assumed to be the best solution (lowest path cost) among all other solutions.
- **Time complexity** - It measures how long an algorithm takes to complete its job.
- **Space Complexity** - The maximum storage space required during the search, as determined by the problem's complexity.

State Space Search



STATE SPACE SEARCH

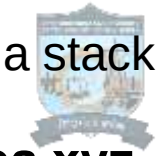
Uninformed/Blind Search

The uninformed search needs domain information, such as proximity or goal location. It works by brute force because it only contains information on traversing the tree and identifying leaf and goal nodes.

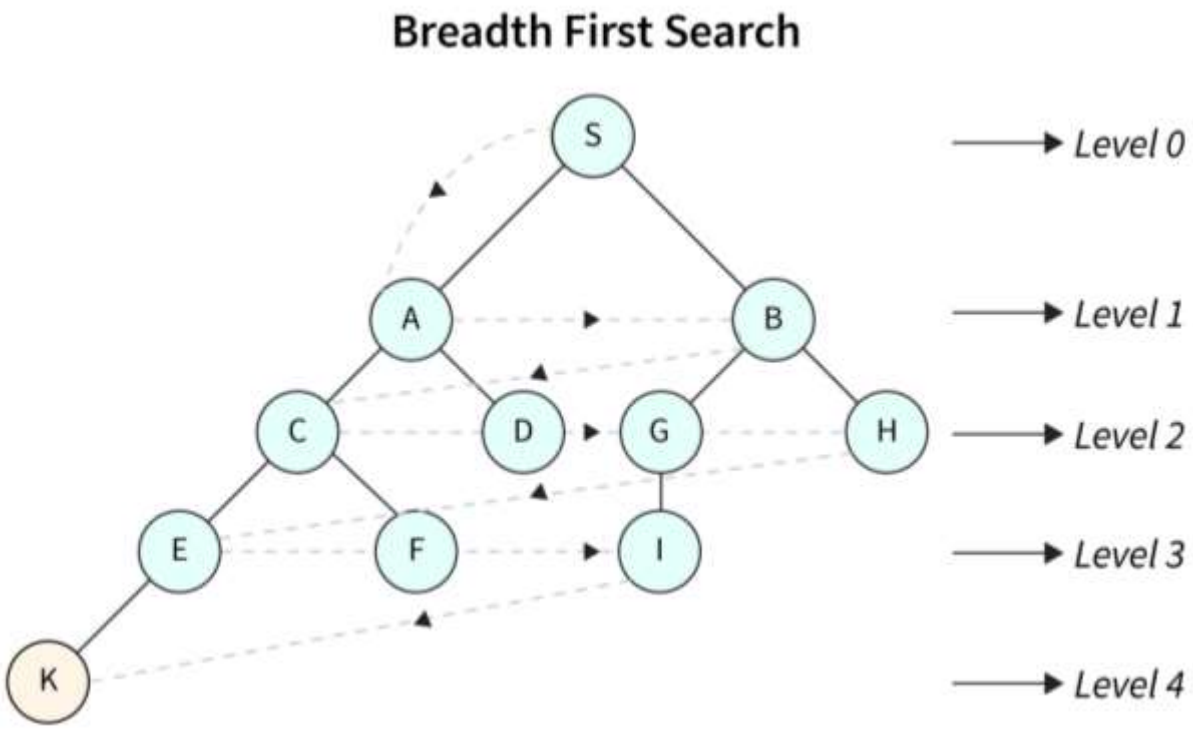
Uninformed search is a method of searching a search tree without knowledge of the search space, such as initial state operators and tests for the objective, and is also known as blind search. It goes through each tree node until it reaches the target node. These algorithms are limited to producing successors and distinguishing between goal and non-goal states.

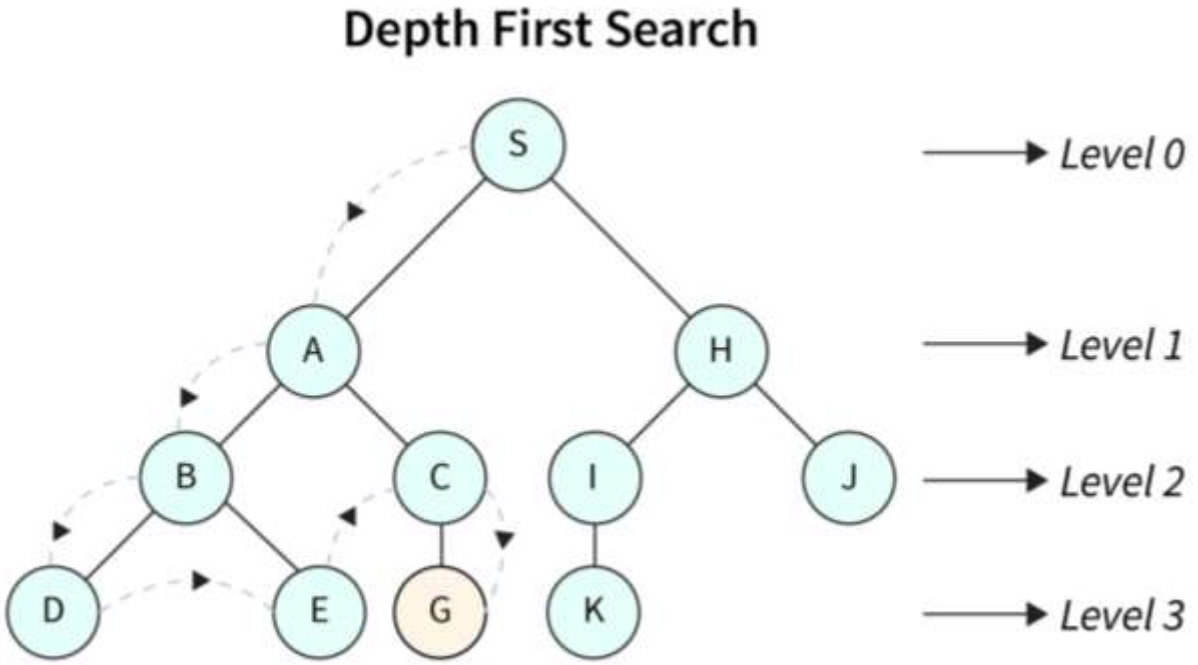
1.Breadth-first search - This is a search method for a graph or tree data structure. It starts at the tree root or searches key and goes through the adjacent nodes in the current depth level before moving on to the nodes in the next depth level. It uses the queue data structure that works on the first in, first out (FIFO) concept. It is a complete algorithm as it returns a solution if a solution exists.

2.Depth-first search - It is also an algorithm used to explore graph or tree data structures. It starts at the root node, as opposed to the breadth-first search. It goes through the branch nodes and then returns. It is implemented using a stack data structure that works on the concept of last in, first out (LIFO).



STATE SPACE SEARCH





STATE SPACE SEARCH

Breadth First Search Algorithm

- **Initialization:**
  1. Start with a designated **"SOURCE"** node and a **"TARGET"** node.
  2. Initialize a queue and add the source node to it.
  3. Maintain a "visited" set or array to keep track of nodes already explored, preventing cycles and redundant processing.
  4. Optionally, maintain a **"PARENT"** dictionary or array to reconstruct the path later, storing the node from which each visited node was reached.
- **Exploration:**
  1. While the queue is not empty:
     a. Dequeue a node (the **"CURRENT"** node).
     b. If the current node is the **TARGET** node, the path is found. Reconstruct the path by backtracking through the **PARENT** pointers from the target to the source.
     c. For each unvisited neighbor of the **CURRENT** node:
        i. Mark the neighbor as visited.
        ii. Set the **CURRENT** node as the neighbor's parent.
        iii. Enqueue the neighbor.
- **Termination:** The algorithm terminates when the **TARGET** node is found or when the queue becomes empty (meaning the target is unreachable from the source).

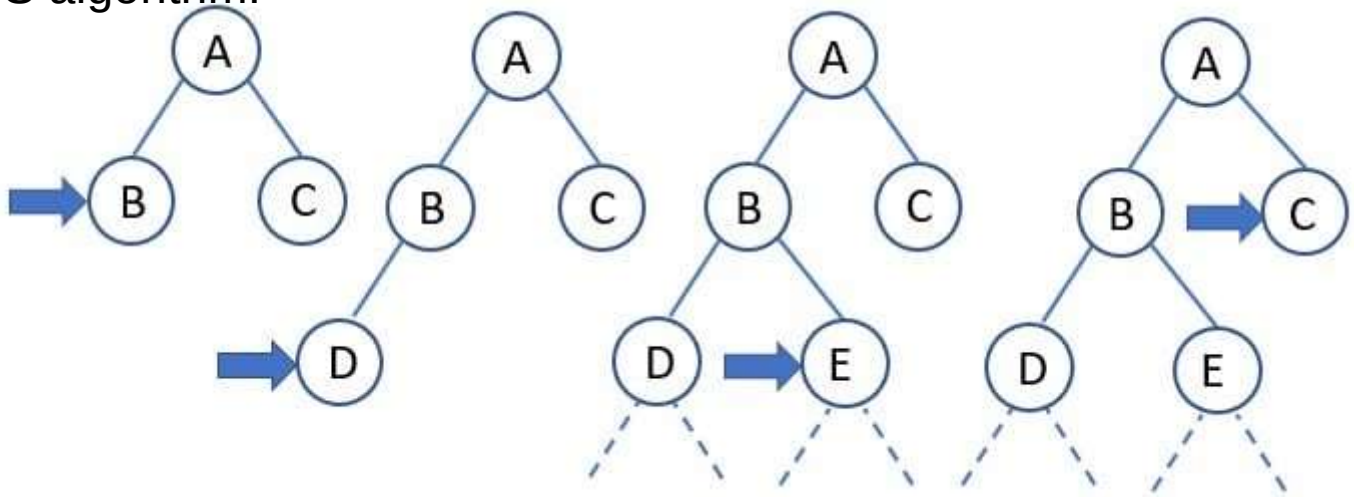
STATE SPACE SEARCH

Depth First Search Algorithm

- **Initialization:**
  1. Start with a designated *source* node and a *target* node.
  2. Initialize a **stack** and push the source node onto it.
  3. Maintain a **visited** set or array to keep track of nodes that have already been explored, preventing infinite loops in cyclic graphs.
  4. Optionally, maintain a **parent** dictionary or array to reconstruct the path later, storing the node from which each visited node was reached.
- **Exploration:**
  1. While the stack is not empty:
     a. Pop a node (the **current** node) from the top of the stack.
     b. If the current node is the target node, the path is found.
        i. Reconstruct the path by backtracking through the parent pointers from the target to the source.
     c. For each unvisited neighbor of the current node:
        i. Mark the neighbor as visited.
        ii. Set the current node as the neighbor's parent.
        iii. Push the neighbor onto the stack.
- **Termination:** The algorithm terminates when the target node is found or when the stack becomes empty (meaning the target is unreachable from the source).

Depth Limited Depth First Search Algorithm

Depth limited search is the new search algorithm for uninformed search. The unbounded tree problem happens to appear in the depth-first search algorithm, and it can be fixed by imposing a boundary or a limit to the depth of the search domain. We will say that this limit as the depth limit, making the DFS search strategy more refined and organized into a finite loop. We denote this limit by *l,* and thus this provides the solution to the infinite path problem that originated earlier in the DFS algorithm.



STATE SPACE SEARCH

Depth Limited Depth First Search Algorithm

Initialization:

1. Start with a designated *source* node and a *target* node.
2. Define a maximum **depth limit** up to which the search will explore.
3. Maintain a **visited** set or array to avoid revisiting nodes in the current path.
4. Optionally, maintain a **parent** dictionary or array to reconstruct the path later.

Exploration (Recursive or Stack-Based):

1. Begin at the source node with the current depth = 0.
2. If the current node is the target node, return success and reconstruct the path using parent pointers.
3. If the current depth equals the depth limit, terminate this branch (do not expand further).
4. For each unvisited neighbor of the current node:
    a. Mark the neighbor as visited.
    b. Set the current node as the neighbor's parent.
    c. Recursively call Depth-Limited Search on the neighbor with depth + 1.

Termination:

1. The search ends when:
    a. The target node is found within the depth limit (success), or
    b. All paths are explored up to the depth limit without finding the target (failure).

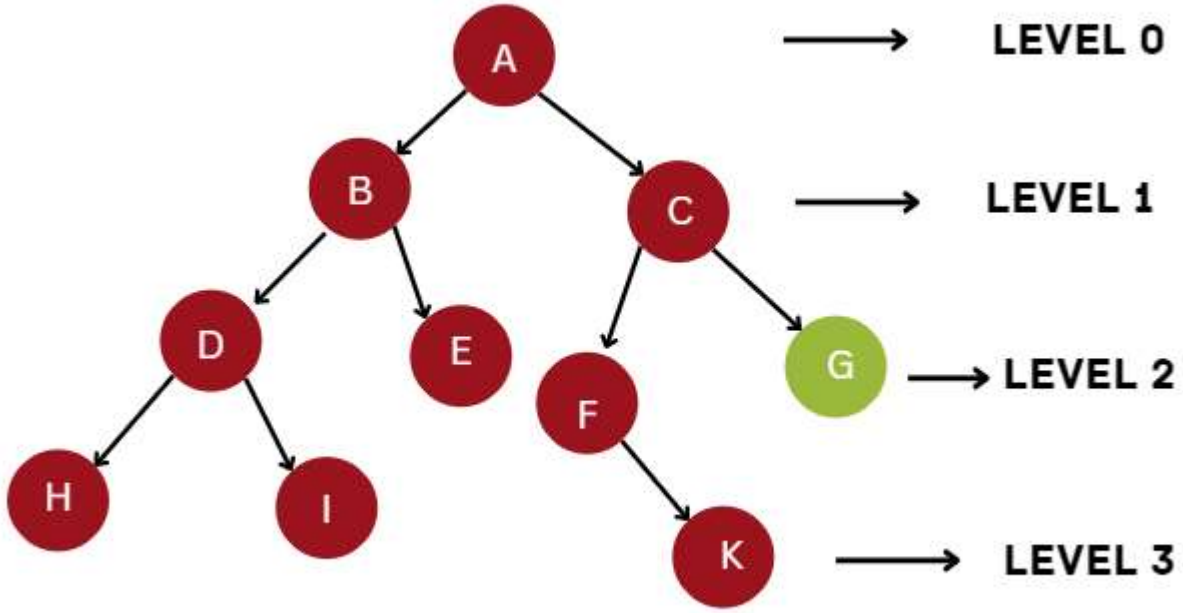
STATE SPACE SEARCH

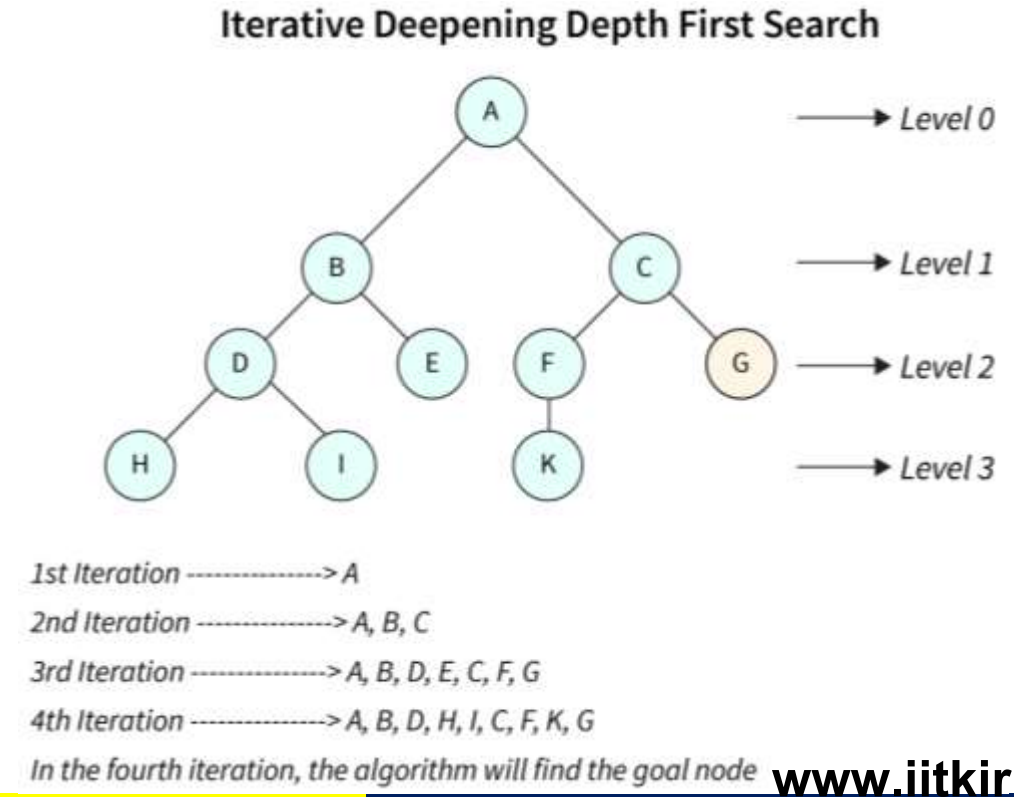
Iterative Deepening Depth-First Search (IDDFS) Algorithm

IDDFS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).

How does IDDFS work?

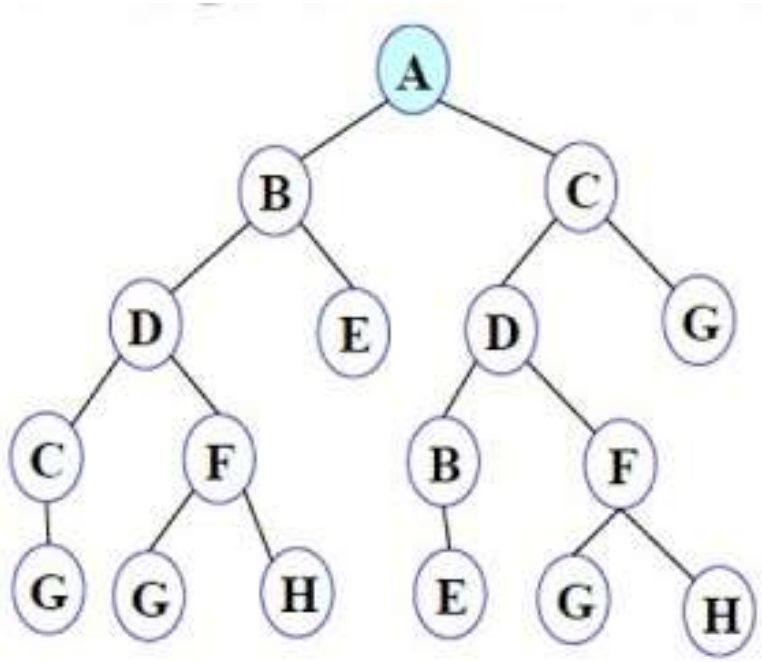
IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically, we do DFS in a BFS fashion.

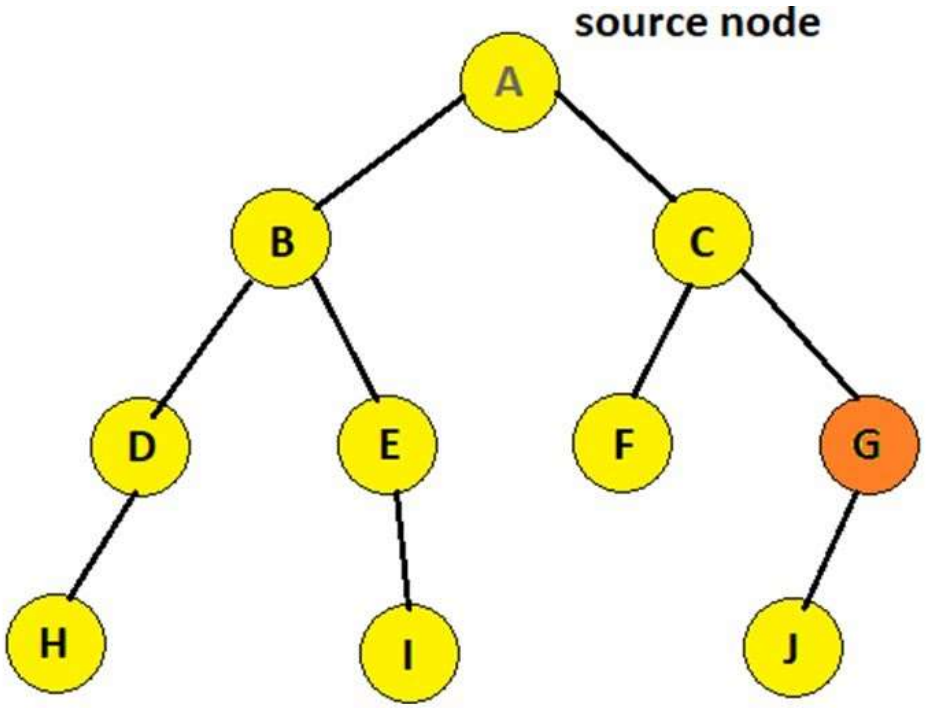




State Space Search

Exercise





STATE SPACE SEARCH

Iterative Deepening Depth-First Search (IDDFS) Algorithm

- **Initialization:**
  1. Start with a designated *source* node and a *target* node.
  2. Set an initial **depth limit** to 0.
  3. Incrementally increase the depth limit by 1 in each iteration until:
     a. The target node is found (success), or
     b. The maximum allowed depth is reached (failure).
- **Exploration:**
  1. For each depth limit d:
     a. Perform a **Depth-Limited Search (DLS)** starting from the source node with the current depth = 0 and limit = d.
     b. If DLS finds the target node, terminate and reconstruct the path.
     c. If DLS reaches the depth limit without success, increase the limit and repeat.
- **Termination:**
  1. The algorithm terminates when:
     a. The target node is found at some depth limit (success), or
     b. All nodes up to the maximum depth have been explored without finding the target (failure).

WATER JUG PROBLEM

The Water Jug Problem is a classic puzzle in artificial intelligence. In this version, there are two jugs: one with a capacity of 4 liters and another with a capacity of 3 liters. Neither jug has any measurement markings. A pump is available to completely fill either jug with water.

The objective is to measure exactly 2 liters of water in the 4-liter jug. Starting with both jugs empty, the task is to determine a sequence of steps that results in exactly 2 liters being poured into the 4-liter jug.

In production rules for the water jug problem, let x denote a 4-litre jug, and y denote a 3-litre jug, i.e. x=0,1,2,3,4 or y=0,1,2,3

- **Start state/Initial State** = (0,0), **Goal state** = (2,n) from any n

Production rules for the water jug problem in AI are as follows:

1.	(x,y) is X<4 -> (4, Y)	Fill the 4-litre jug
2.	(x, y) if Y<3 -> (x, 3)	Fill the 3-litre jug
3.	(x, y) if x>0 -> (x-d, d)	Pour some water from a 4-litre jug
4.	(x, y) if Y>0 -> (d, y-d)	Pour some water from a 3-litre jug
5.	(x, y) if x>0 -> (0, y)	Empty 4-litre jug on the ground
6.	(x, y) if y>0 -> (x,0)	Empty 3-litre jug on the ground
7.	(x, y) if X+Y >= 4 and y>0 -> (4, y-(4-x))	Pour water from a 3-litre jug into a 4-litre jug until it is full
8.	(x, y) if X+Y>=3 and x>0 -> (x-(3-y), 3))	Pour water from a 3-litre jug into a 4-litre jug until it is full
9.	(x, y) if X+Y <=4 and y>0 -> (x+y, 0)	Pour all the water from a 3-litre jug into a 4-litre jug
10.	(x, y) if X+Y<=3 and x>0 -> (0, x+y)	Pour all the water from a 4-litre jug into a 3-litre jug
11.	(0, 2) -> (2, 0)	Pour 2-litre water from 3-litre jug into 4-litre jug
12.	(2, Y) -> (0, y)	Empty 2-litre in the 4-litre jug on the ground.

WATER JUG PROBLEM

Step	State (x, y)	Applied Rule	Description
0	(0, 0)	—	Initial state
1	(0, 3)	Rule 2	Fill 3L jug
2	(3, 0)	Rule 9	Pour 3L into 4L
3	(3, 3)	Rule 2	Fill 3L jug
4	(4, 2)	Rule 7	Pour from 3L → 4L until full
5	(0, 2)	Rule 5	Empty 4L jug
6	(2, 0)	Rule 11	Pour 2L into 4L jug **(Goal)**

BFS State-Space Tree

- **Level 0 (Initial State)**
  (0, 0)
- **Level 1 (Apply Rule 1 & 2)**
  (4, 0) – Fill 4L jug
  (0, 3) – Fill 3L jug
- **Level 2 (From (0, 3) using Rule 9)**
  (3, 0) – Pour 3L → 4L
- **Level 3 (From (3, 0) using Rule 2)**
  (3, 3) – Fill 3L jug
- **Level 4 (From (3, 3) using Rule 7)**
  (4, 2) – Pour 3L → 4L until full
- **Level 5 (From (4, 2) using Rule 5)**
  (0, 2) – Empty 4L jug
- **Level 6 (From (0, 2) using Rule 11)**
  (2, 0) – Pour 2L → 4L **[Goal**

STATE SPACE SEARCH

Informed/Heuristic Search

Informed search algorithms are a class of algorithms in artificial intelligence that use heuristic functions to guide the search process. A heuristic is a function that estimates the cost of reaching the goal from a given node, providing additional information that helps the algorithm make smarter decisions.

Role of Heuristics

Heuristics play a crucial role in informed search algorithms. They help prioritize which nodes or paths the algorithm should explore first by estimating how close a node is to the goal. This dramatically reduces the number of states explored, making the search process more efficient.

Comparison with Uninformed Search

In contrast to informed search, **uninformed search algorithms** like breadth-first or depth-first search explore the search space without any additional information, often leading to longer search times and inefficient exploration. For example, breadth-first search explores all possible states level by level, which can be highly time-consuming in large search spaces.

STATE SPACE SEARCH

Key Characteristics of Informed Search Algorithms

1. **Heuristic Function**: Informed search algorithms use a heuristic function ***h(n)*** that provides an estimate of the minimal cost from node n to the goal. This function helps the algorithm to prioritize which nodes to explore first based on their potential to lead to an optimal solution.

2. **Efficiency**: By focusing on more promising paths, informed search algorithms often find solutions more quickly than uninformed methods, especially in large or complex search spaces.

3. **Optimality and Completeness**: Depending on the heuristic used, informed search algorithms can be both optimal and complete. An algorithm is complete if it is guaranteed to find a solution if one exists, and it is optimal if it always finds the best solution. For instance, the A* search algorithm is both complete and optimal when the heuristic function is admissible (i.e., it never overestimates the true cost).

4. **Admissibility and Consistency of Heuristics:** An admissible heuristic guarantees that the algorithm finds the optimal solution. A heuristic is admissible if it never overestimates the true cost to reach the goal.

1. **Hill climbing Search**
2. **Greedy Best-First Search**
3. **A* Algorithm**
4. **AO* Algorithm**

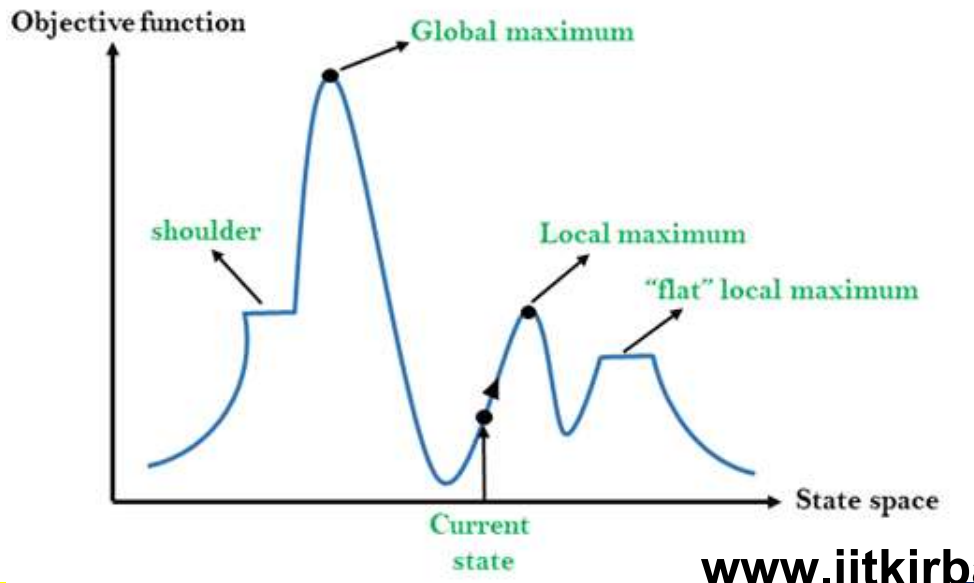
HILL CLIMBING

Hill climbing is a widely used **optimization algorithm** in **Artificial Intelligence (AI)** that helps find the best possible solution to a given problem. As part of the **local search algorithms** family, it is often applied to **optimization problems** where the goal is to identify the optimal solution from a set of potential candidates.

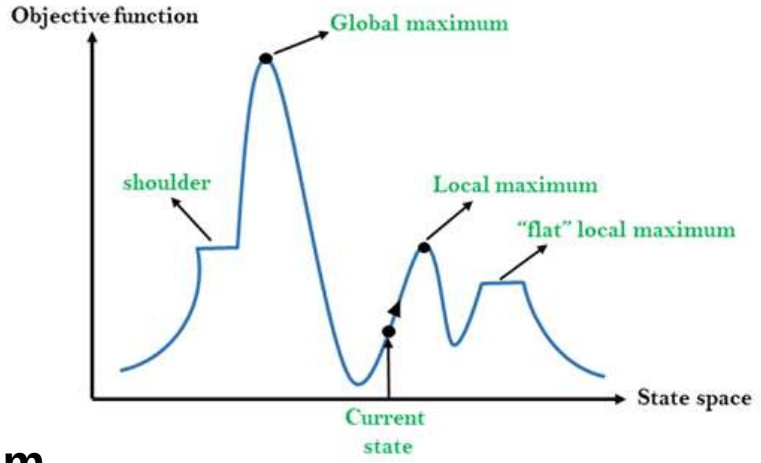
In the **Hill Climbing algorithm**, the **state-space diagram** is a visual representation of all possible states the search algorithm can reach, plotted against the values of the **objective function** (the function we aim to maximize).
In the state-space diagram:

- **X-axis**: Represents the state space, which includes all the possible states or configurations that the algorithm can reach.
- **Y-axis**: Represents the values of the **objective function** corresponding to each state.

The optimal solution in the state-space diagram is represented by the state where the **objective function** reaches its maximum value, also known as the **global maximum**.



HILL CLIMBING



Key Regions in the State-Space Diagram

1.Local Maximum: A local maximum is a state better than its neighbors but not the best overall. While its objective function value is higher than nearby states, a global maximum may still exist.

2.Global Maximum: The global maximum is the best state in the state-space diagram, where the objective function achieves its highest value. This is the optimal solution the algorithm seeks.

3.Plateau/Flat Local Maximum: A plateau is a flat region where neighboring states have the same objective function value, making it difficult for the algorithm to decide on the best direction to move.

4.Ridge: A ridge is a higher region with a slope, which can look like a peak. This may cause the algorithm to stop prematurely, missing better solutions nearby.

5.Current State: The current state refers to the algorithm's position in the state-space diagram during its search for the optimal solution.

6.Shoulder: A shoulder is a plateau with an uphill edge, allowing the algorithm to move toward better solutions if it continues searching beyond the plateau.

HILL CLIMBING - TYPES

Types of Hill Climbing

1. **Simple Hill Climbing:** This is the most basic form of hill climbing. The algorithm evaluates each neighboring state in sequence and moves to the first neighbor that improves the objective function.
   - **Pros**: Simple and easy to implement.
   - **Cons**: Can get stuck in local optima easily and may require many iterations to find a better solution.

Steps:

1. **Start with an Initial Solution**: Choose an initial state (solution) randomly or based on some heuristic.
2. **Evaluate the Initial Solution**: Compute the value of the objective function for the current state.
3. **Generate Neighbors**: Generate the neighboring states of the current state.
4. **Evaluate Neighbors**: Evaluate each neighbor to see if it improves the objective function.
5. **Select the First Better Neighbor**: Move to the first neighbor that has a better (higher or lower, depending on the problem) objective function value.
6. **Repeat**: Repeat steps 3-5 until no improvement is found (i.e., no neighbor is better than the current state).

HILL CLIMBING - TYPES

2. **Steepest-Ascent Hill Climbing (Gradient Ascent/Descent):** Evaluates all neighbors and moves to the neighbor with the highest improvement (or lowest cost).
   - **Pros**: More likely to find a better solution than simple hill climbing since it considers all possible moves.
   - **Cons**: More computationally expensive as it evaluates all neighbors before making a move.

Steps:

1. **Start with an Initial Solution**: Choose an initial state (solution) randomly or based on some heuristic.
2. **Evaluate the Initial Solution**: Compute the value of the objective function for the current state.
3. **Generate Neighbors**: Generate the neighboring states of the current state.
4. **Evaluate All Neighbors**: Evaluate all neighbors to determine the one with the best (highest or lowest) objective function value.
5. **Select the Best Neighbor**: Move to the neighbor with the best objective function value.
6. **Repeat**: Repeat steps 3-5 until no neighbor improves the objective function.

HILL CLIMBING - TYPES

3. **Stochastic Hill Climbing:** Selects a random neighbor and moves to it if it improves the objective function. The randomness helps in exploring the search space more broadly.
   - **Pros**: Can escape local optima more effectively than simple hill climbing.
   - **Cons**: Less predictable and may require more iterations to converge.

Steps:

1. **Start with an Initial Solution**: Choose an initial state (solution) randomly or based on some heuristic.
2. **Evaluate the Initial Solution**: Compute the value of the objective function for the current state.
3. **Generate a Random Neighbor**: Generate a random neighboring state of the current state.
4. **Evaluate the Neighbor**: Compute the value of the objective function for the random neighbor.
5. **Accept or Reject the Neighbor**: Move to the neighbor if it has a better (higher or lower) objective function value.
6. **Repeat**: Repeat steps 3-5 until no further improvement is found.

HILL CLIMBING – LIMITATION & CHALLENGES

1. **Local Optima:**
   - Hill Climbing is a local search algorithm, which means it tends to get stuck in local optima solutions that are better than their immediate neighbors but not necessarily the global optimum.
   - This challenge arises because Hill Climbing always moves to a better neighboring solution, even if there's a much better solution further away. It lacks the ability to explore beyond the current local neighborhood.
2. **Sensitivity to Initial Conditions:** The effectiveness of Hill Climbing is highly dependent on the choice of the initial solution. Different initial solutions can lead to entirely different local optima or even failed convergence.
3. **Lack of Global Exploration:** Hill Climbing is inherently focused on improving the current solution and may miss out on better solutions in distant parts of the search space. It lacks a mechanism for global exploration, which is crucial for finding the global optimum.
4. **Plateau Problem:** A **plateau** is a flat region in the search space where all neighboring states have the same value. This makes it difficult for the algorithm to choose the best direction to move forward.
5. **Ridge Problem:** A **ridge** is a region where movement in all possible directions seems to lead downward, resembling a peak. As a result, the Hill Climbing algorithm may stop prematurely, believing it has reached the optimal solution when, in fact, better solutions exist.

BEST FIRST SEARCH

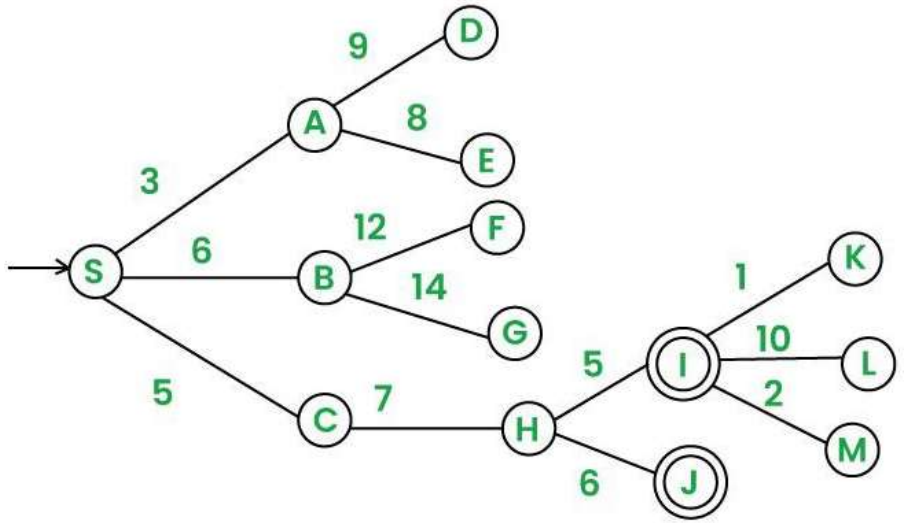
Best First Search is a heuristic search algorithm that selects the most promising node for expansion based on an evaluation function. It prioritizes nodes in the search space using a heuristic to estimate their potential. By iteratively choosing the most promising node, it aims to efficiently navigate towards the goal state, making it particularly effective for optimization problems.

$f(n)$ **= Evaluating Function (Path Cost from Node A to Node B)**

Best First Search (BFS) follows a graph by using a priority queue and heuristics. It keeps an 'Open' list for nodes that need exploring and a 'Closed' list for those already checked. Here's how it operates:

1. Create 2 empty lists: **OPEN and CLOSED**
2. Start from the initial node (say **N**) and put it in the 'ordered' OPEN list
3. Repeat the next steps until the GOAL node is reached
    a. If the **OPEN** list is empty, then **EXIT** the loop returning 'False'
    b. Select the first/top node (say **N**) in the **OPEN** list and move it to the **CLOSED** list. Also, capture the information of the parent node
    c. If N is a **GOAL** node, then move the node to the **CLOSED** list and exit the loop returning '**True**'. The solution can be found by backtracking the path
    d. If N is not the **GOAL** node, expand node N to generate the '**immediate**' next nodes linked to node N and add all those to the **OPEN** list
    e. Reorder the nodes in the OPEN list in ascending order according to an evaluation function **f(n)**

BEST FIRST SEARCH

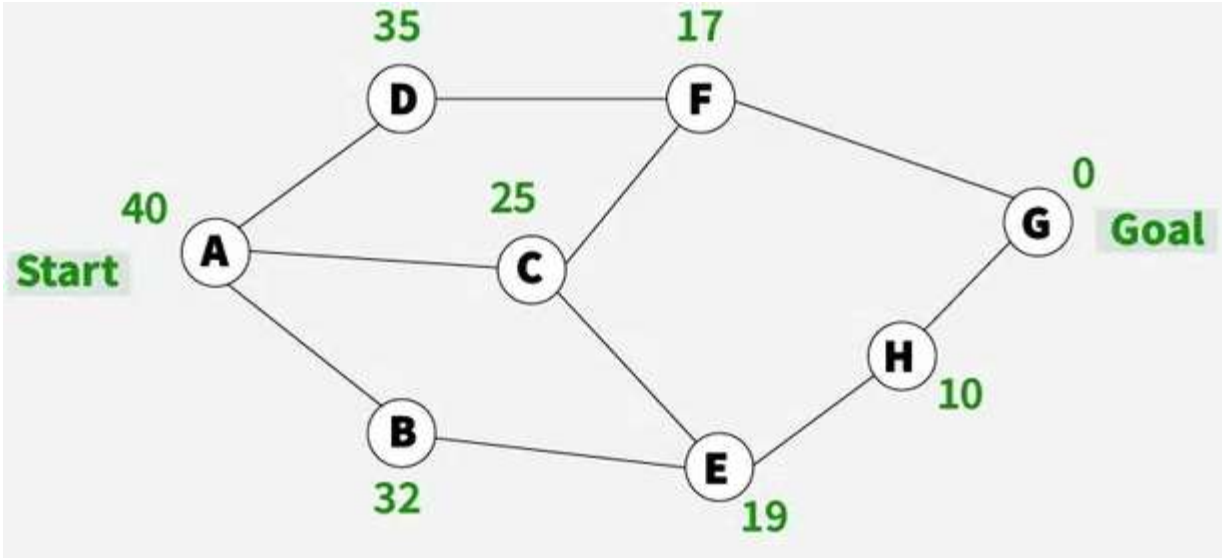


Greedy Best First Search

Evaluating Function

$$f(n) = h(n)$$

h(n) = Heuristic Function



A* ALGORITHM

A* (A-Star) is a popular choice for pathfinding because it combines the strengths of Dijkstra's algorithm and Greedy Best-First Search, making it efficient and optimal. It uses a heuristic function to evaluate paths by balancing the cost to reach a node **(g-cost [*g(n)*])** and the estimated cost to the goal (h-cost **[*h(n)*]**).

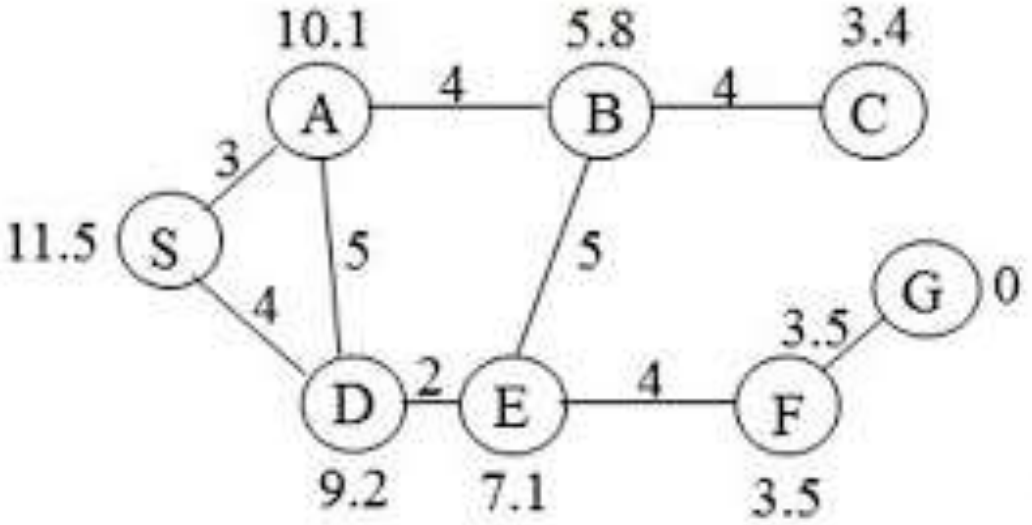
A* uses **two important** parameters to find the cost of a path:

1. ***g*(*n*):** Actual cost of reaching node *n* from the start node. This is the accumulated cost of the path from the start node to node *n*.

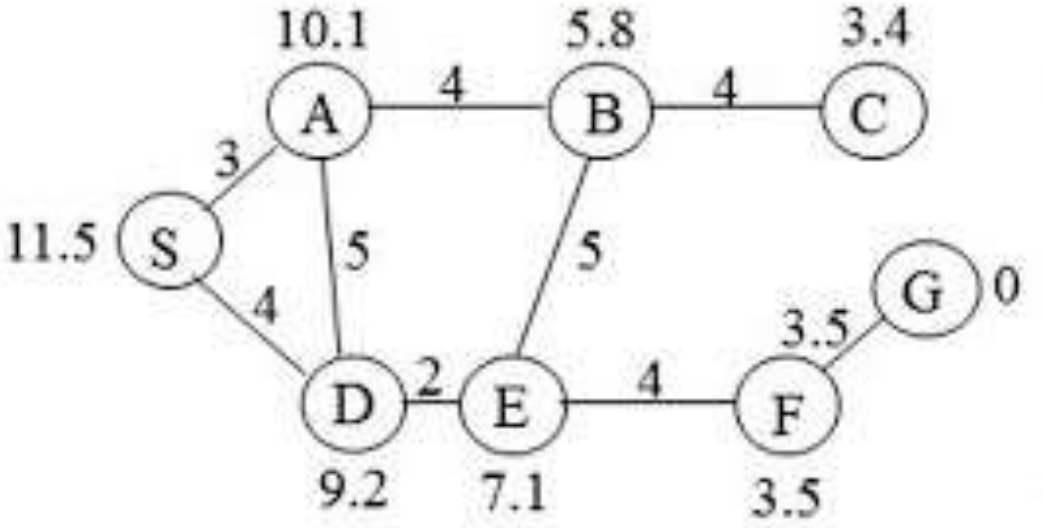
2. ***h*(*n*):** The heuristic finds of the cost to reach the goal from node *n*. This is a weighted guess about how much further it will take to reach the goal.

The function, ***f*(*n*)=*g*(*n*)+*h*(*n*)** is the total estimated cost of the cheapest solution through node *n*. This function combines the path cost so far and the heuristic cost to estimate the total cost guiding the search more efficiently.

A* ALGORITHM



A* ALGORITHM



Find the most cost-effective path to reach the final state from initial state using A* Algorithm. Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

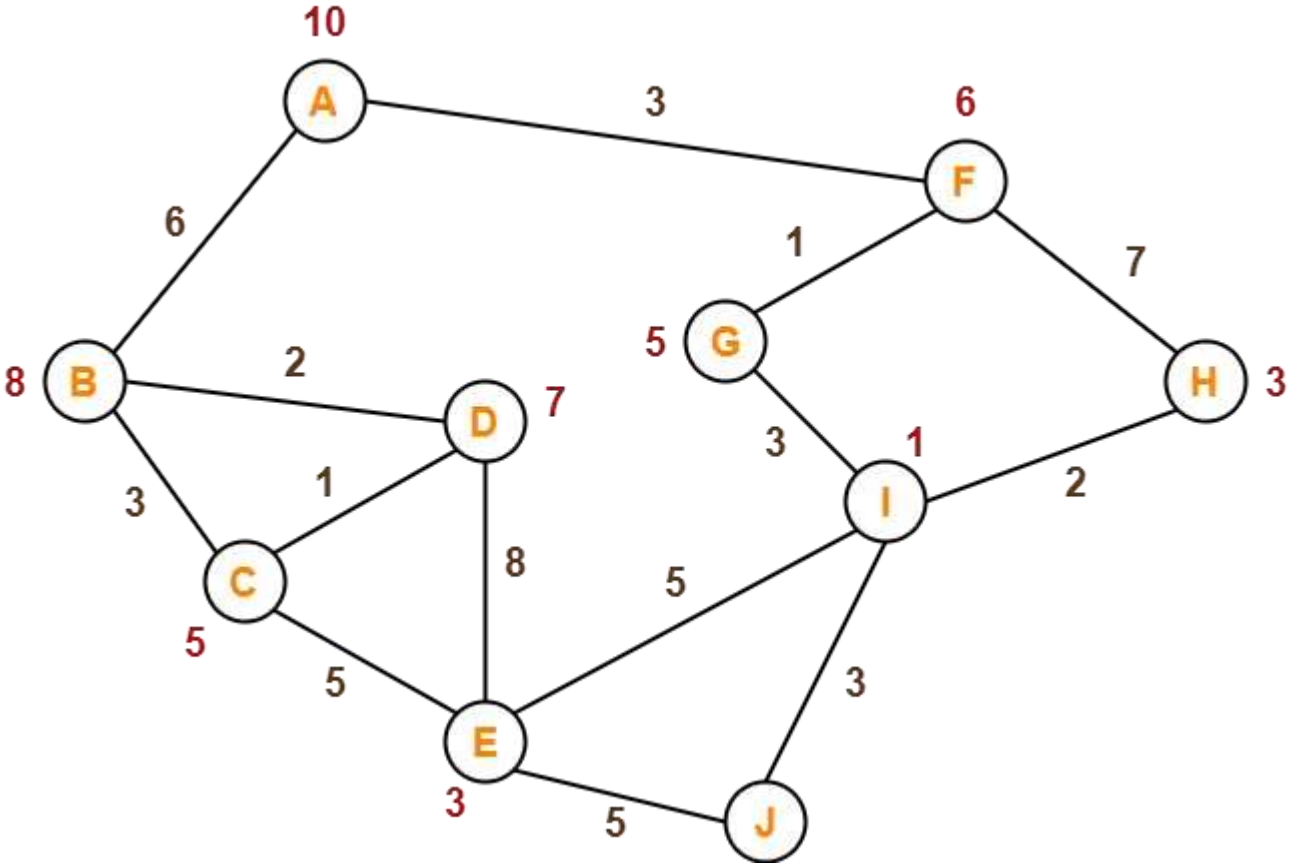
2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

Final State

Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.



ADMISSIBILITY OF A* ALGORITHM

The cost of reaching the goal state is assessed using an admissible heuristic in an informed search algorithm, however, if we need to discover a solution to the problem, the estimated cost must be lower than or equal to the true cost of reaching the goal state. The algorithm employs the allowable heuristic to determine the best-estimated route from the current node to the objective state.

The evaluation function in **A*** looks like this:

f(n) = g(n) + h(n)

f(n) = Actual cost + Estimated cost

here,

n = current node.

f(n) = evaluation function.

g(n) = the cost from the initial node to the current node.

h(n) = estimated cost from the current node to the goal state.

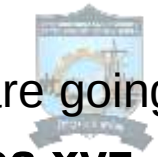
Conditions:

h(n) ≤ h*(n) ∴ Underestimation

h(n) ≥ h*(n) ∴ Overestimation

Key Points:

- The heuristic function h(n) is admissible if **h(n)** is **never larger** than **h*(n)** or if **h(n)** is always less or equal to the true value.
- If **A*** employs an admissible heuristic and **h(goal)=0**, then we can argue that **A* is admissible.**
- If the heuristic function is constantly tuned to be low with respect to the true cost, i.e. **h(n) ≤ h*(n)**, then you are going to get an optimal solution of 100%



ADMISSIBILITY OF A* ALGORITHM

Case 1:

Let's suppose, you are going to purchase shoes and shoes have a price of **$1000**. Before making the purchase, you estimated that the shoes will be worth **$1200**, When you went to the store, the shopkeeper informed you that the shoe's actual price was **$1000**, which is less than your estimated value, indicating that you had overestimated their value by **$200**. so this is the case of **Overestimation.**

1200 > 1000

i.e. h(n) ≥ h*(n) ∴ Overestimation

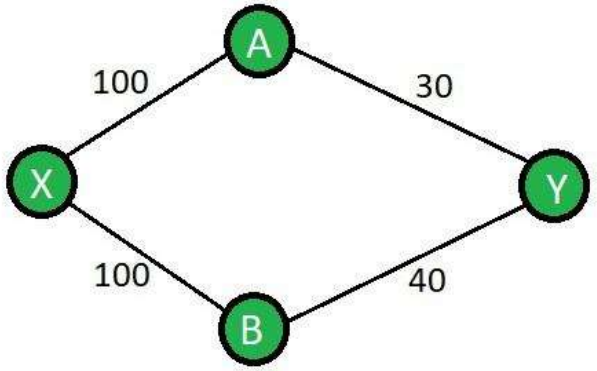
Case 2:

Similar to the last situation, you are planning to buy a pair of shoes. This time, you estimate the shoe value to be **$800.** However, when you arrive at the store, the shopkeeper informs you that the shoes' true price is **$1000,** which is higher than your estimate. Indicating that you had underestimated their value by **$200**. In this situation, **Underestimation** has occurred.

800 < 1000

i.e. h(n) ≤ h*(n) ∴ Underestimation

ADMISSIBILITY OF A* ALGORITHM



Case 1: Overestimation

H(A)= 60, Estimated values i.e. h(n)]
H(B)= 50
So, using A* equation, **f(n) = G(n) + h(n)**
by putting values

f(A) = 100 + 60 = 160
f(B) = 100 + 50 = 150

by comparing f(A) & f(B), f(A) > f(B) so choose path of B node and apply A* equation again

f(Y) = g(Y) + h(Y) [here h(Y) is 0, Goal state]
= 140 + 0 [g(Y)=100+40=140 this is actual cost i.e. h*(B)]
= **140**

Case 2: Underestimation

H(A) = 20 [This are estimated values i.e. h(n)]
H(B) = 10
So, using A* equation, **f(n) = G(n) + h(n)**
by putting values

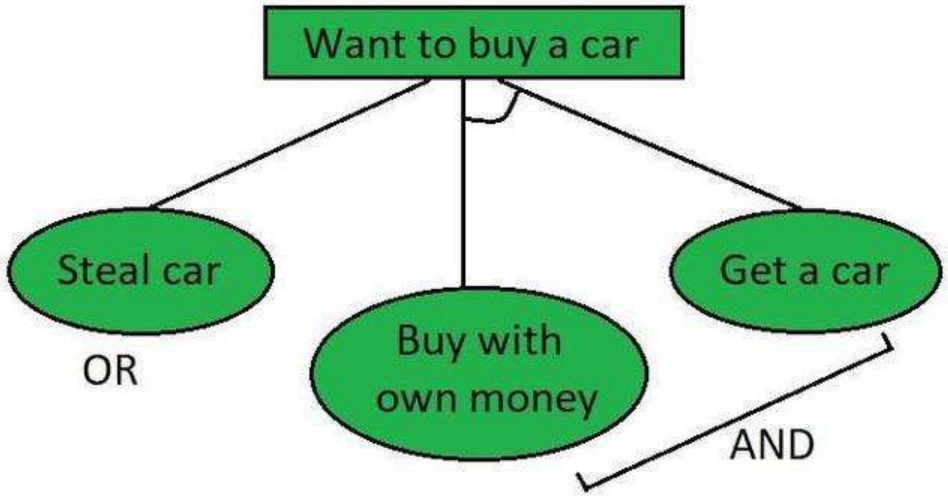
f(A) = 100 + 20 = 120
f(B) = 100 + 10 = 110, by comparing f(A) & f(B), f(A) > f(B), so choose path of B node and apply A* equation again
f(Y) = g(Y) + h(Y) [here h(Y) is 0, because it is goal state]
= 140 + 0 [g(Y) = 100 + 40 = 140 this is actual cost i.e. h*(B)]
= **140**

f(Y) = g(Y) + h(Y)
= 130 + 0 = 130

AO* ALGORITHM

The AO* algorithm is an advanced search algorithm utilized in artificial intelligence, particularly in problem-solving and decision-making contexts. It is an extension of the A* algorithm, designed to handle more complex problems that require handling multiple paths and making decisions at each node.

The ***AO* algorithm*** **(short for "And-Or Star")** is a powerful **best-first search** method used to solve problems that can be represented as a **directed acyclic graph (DAG)**. Unlike traditional algorithms like **A***, which explore a single path, the AO* algorithm evaluates multiple paths simultaneously.



AO* Algorithm - Working Principles

The AO* algorithm works by utilizing a tree structure where each node represents a state in the problem space. The key components of the algorithm are:

Node Types

- **AND Nodes**: Represent states where all child nodes must be satisfied to achieve a goal. If a task requires multiple conditions to be met, it would be represented as an AND node.
- **OR Nodes**: Represent states where at least one child node must be satisfied to achieve a goal. This type is useful in scenarios where multiple paths can lead to a solution.

Heuristic Function

The algorithm employs a heuristic function, similar to A*, to estimate the cost to reach the goal from any given node. This function helps in determining the most promising paths to explore. The heuristic function $h(n)$ estimates the cost to reach the goal from node n: $h(n)$=estimated cost to reach the goal from node

- **For OR Nodes**: The algorithm considers the lowest cost among the child nodes. The cost for an OR node can be expressed as:

$C(n)=\min\{C(c1),C(c2),\ldots,C(ck)\}$

- **For AND Nodes**: The algorithm computes the cost of all child nodes and selects the maximum cost, as all conditions must be met. The cost for an AND node can be expressed as:

$C(n)=\max\{C(c1),C(c2),\ldots,C(ck)\}$

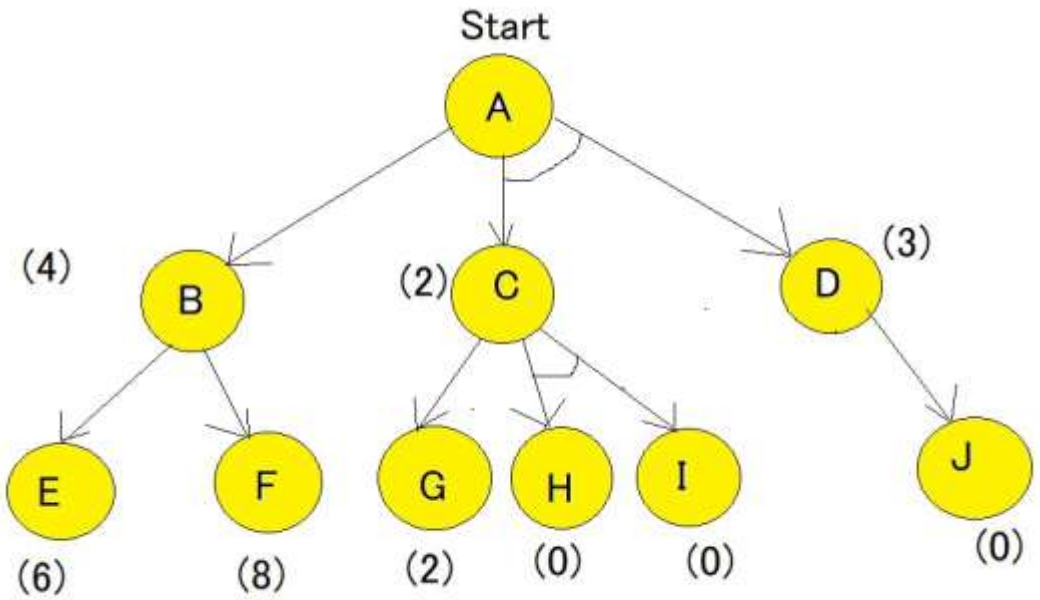
Total Estimated Cost: $f(n)=C(n)+h(n)$

- $C(n)$ is the actual cost to reach node n from the start node.
- $h(n)$ is the estimated cost from node n to the goal.

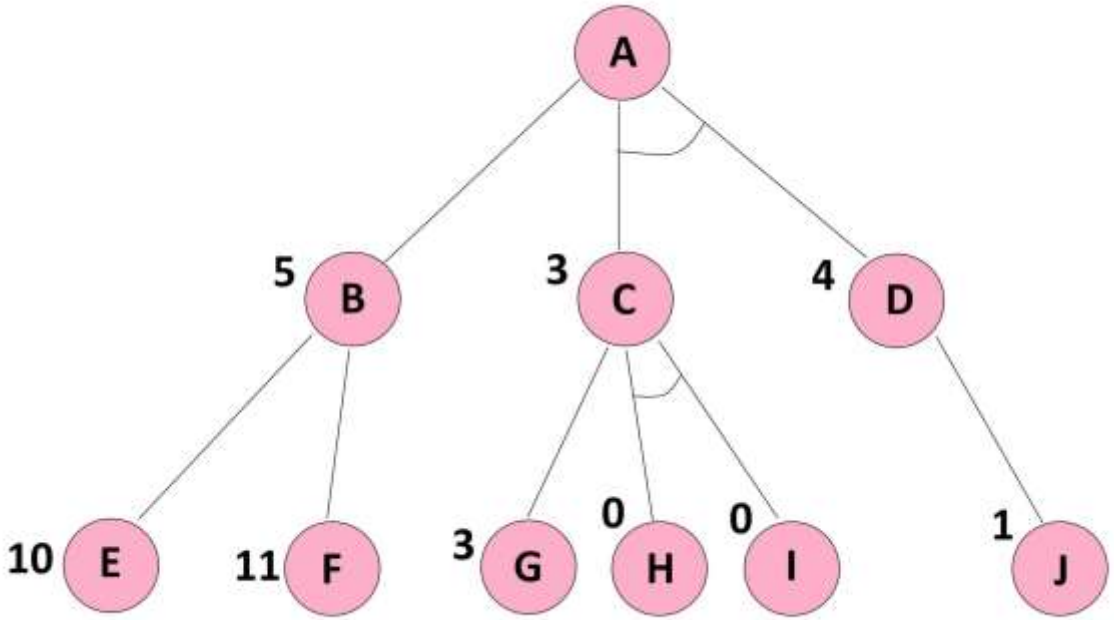
AO* ALGORITHM

1. Initialise the graph to start node
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved
3. Pick any of these nodes and expand it and if it has no successors call this value FUTILITY otherwise calculate only f' for each of the successors.
4. If f' is 0 then mark the node as SOLVED
5. Change the value of f' for the newly created node to reflect its successors by back propagation.
6. Wherever possible use the most promising routes and if a node is marked as SOLVED then mark the parent node as SOLVED.
7. If starting node is SOLVED or value greater than FUTILITY, stop, else repeat from 2.

AO* ALGORITHM - WORKING PRINCIPLES



AO* Algorithm - Working Principles



MIN-MAX ALGORITHM

Min-Max algorithm is a decision-making algorithm used in artificial intelligence, particularly in game theory and computer games. It is designed to minimize the possible loss in a worst-case scenario (hence "min") and maximize the potential gain (therefore "max").

Working of Min-Max Process in AI

Min-Max algorithm involves two players: the maximizer and the minimizer, each aiming to optimize their own outcomes.

Players Involved

Maximizing Player (Max):

- Aims to maximize their score or utility value.
- Chooses the move that leads to the highest possible utility value, assuming the opponent will play optimally.

Minimizing Player (Min):

- Aims to minimize the maximizer's score or utility value.
- Selects the move that results in the lowest possible utility value for the maximizer, assuming the opponent will play optimally.

Min-Max Algorithm

The Min-Max algorithm involves several key steps, executed recursively until the optimal move is determined. Here is a step-by-step breakdown:

Step 1: Generate the Game Tree

- **Objective**: Create a tree structure representing all possible moves from the current game state.
- **Details**: Each node represents a game state, and each edge represents a possible move.

Step 2: Evaluate Terminal States

- **Objective**: Assign utility values to the terminal nodes of the game tree.
- **Details**: These values represent the outcome of the game (win, lose, or draw).

Step 3: Propagate Utility Values Upwards

- **Objective**: Starting from the terminal nodes, propagate the utility values upwards through the tree.
- **Details**: For each non-terminal node:
  - If it's the maximizing player's turn, select the maximum value from the child nodes.
  - If it's the minimizing player's turn, select the minimum value from the child nodes.

Step 4: Select Optimal Move

- **Objective**: At the root of the game tree, the maximizing player selects the move that leads to the highest utility value.

MIN-MAX ALGORITHM

Min-Max Formula

The Min-Max value of a node in the game tree is calculated using the following recursive formulas:

1. Maximizing Player's Turn:

$$Max(s) = max_{a \in A(s)} Max(Result(s, a))$$

Here:

- $Max(s)$is the maximum value the maximizing player can achieve from state s.
- $A(s)$ is the set of all possible actions from state s.
- $Result(s, a)$ is the resulting state from taking action a*a* in state s.
- $Min(Result(s, a))$ is the value for the minimizing player from the resulting state.

Minimizing Player's Turn:

$$Min(s) = min_{a \in A(s)} Max(Result(s, a))$$

Here:

- $Min(s)$ is the minimum value the minimizing player can achieve from state s.
- The other terms are similar to those defined above.

Terminal States

For terminal states, the utility value is directly assigned:

$$Utility(s) = \begin{cases} 1 & if\ the\ maximizing\ player\ wins\ from\ state\ s \\ 0 & if\ the\ game\ is\ draw\ from\ state\ s \\ -1 & if\ the\ minimizing\ player\ wins\ from\ state\ s \end{cases}$$

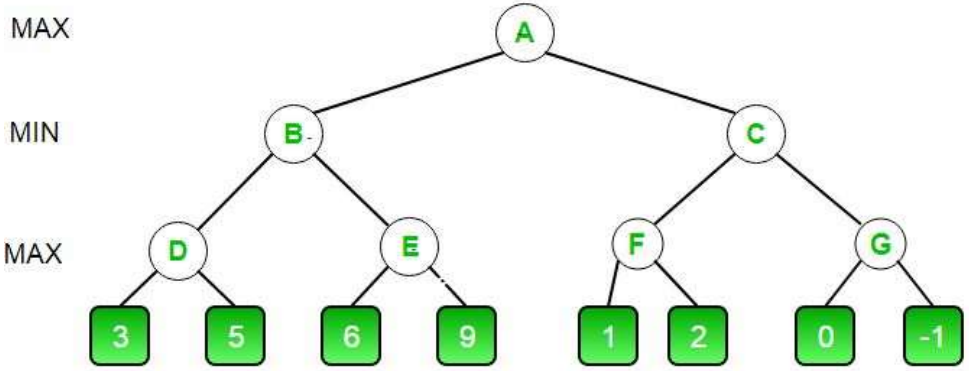
MIN-MAX ALGORITHM

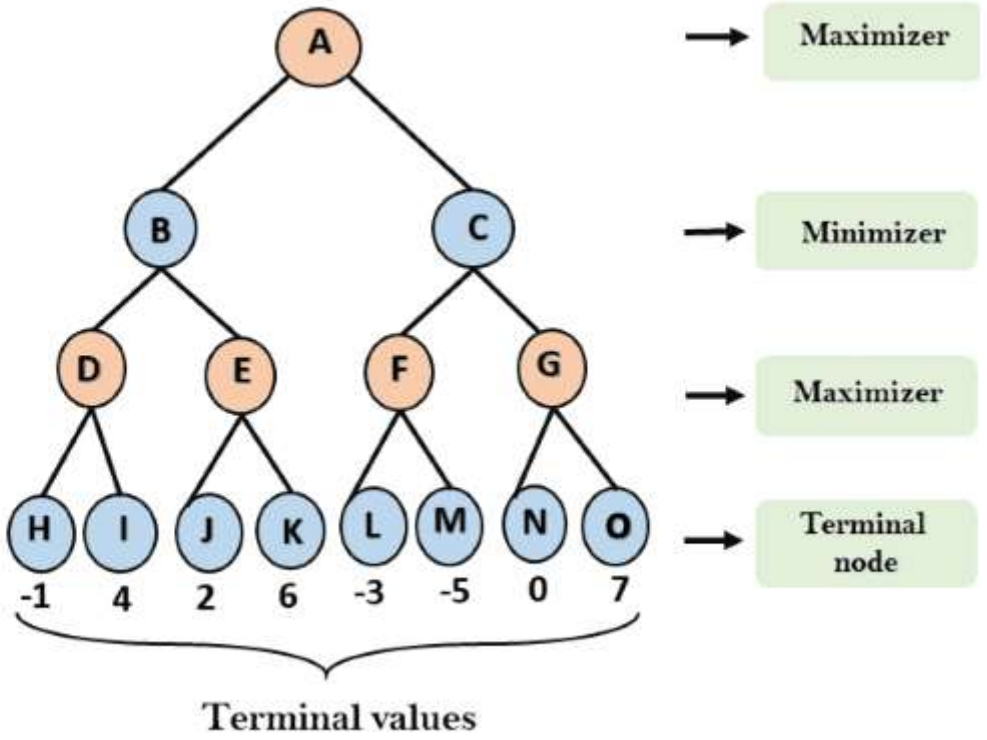
Consider a simple game where the utility values of terminal states are given. To illustrate the Min-Max calculations:

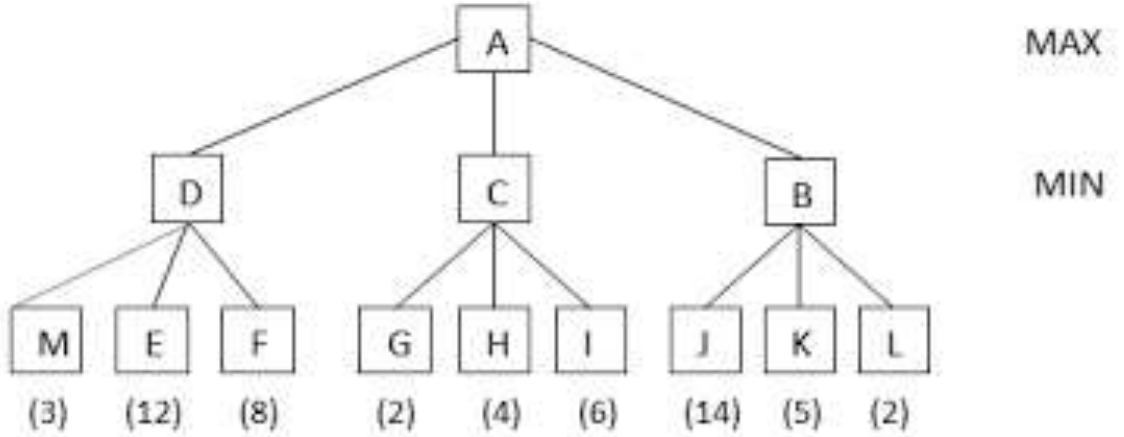
1. Start from the terminal states and calculate the utility values.
2. Propagate these values up the tree using the Min-Max formulas.

For example, if the terminal states have utility values $U_1, U_2, \ldots, U_n$ then:

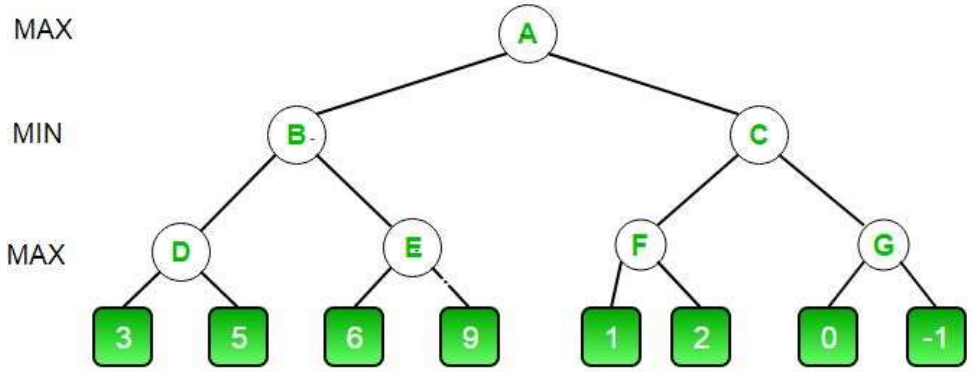
- For the maximizing player's node: $Max(s) = \max(U_1, U_2, \ldots, U_n)$
- For the minimizing player's node: $Min(s) = \min(U_1, U_2, \ldots, U_n)$

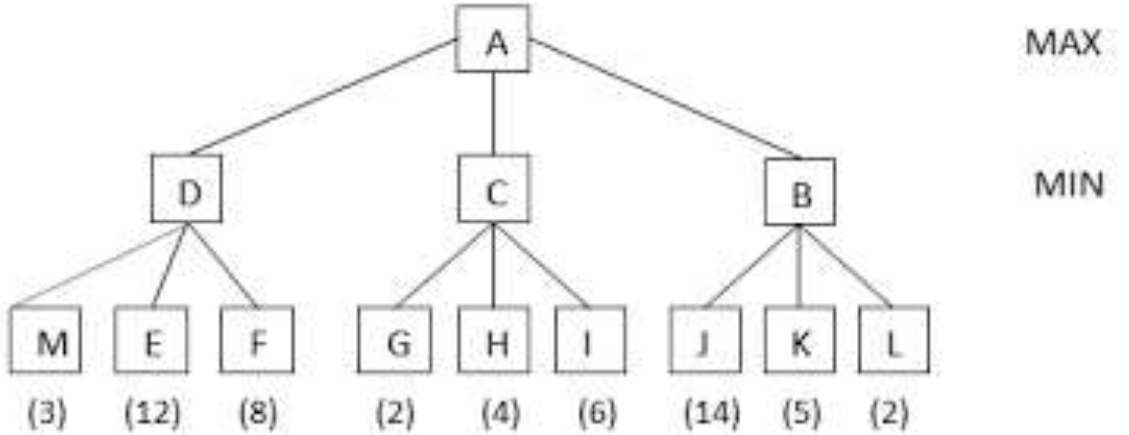






MIN-MAX ALGORITHM





ALPHA BETA PRUNING

The key idea behind Alpha Beta Pruning is to avoid evaluating branches of the game tree that cannot influence the final decision based on the values already discovered during the search. It achieves this using two values: **Alpha** and **Beta**.

The critical points of Alpha Beta Pruning in AI are as follows.

The initialization of the parameters

- $Alpha(\alpha)$ is initialized with $-\infty$.
- $Beta(\beta)$ is initialized with $+\infty$.

Updating the parameters

- $Alpha(\alpha)$ is updated only by the maximizer at its turn.
- $Beta(\beta)$is updated only by the minimizer at its turn.

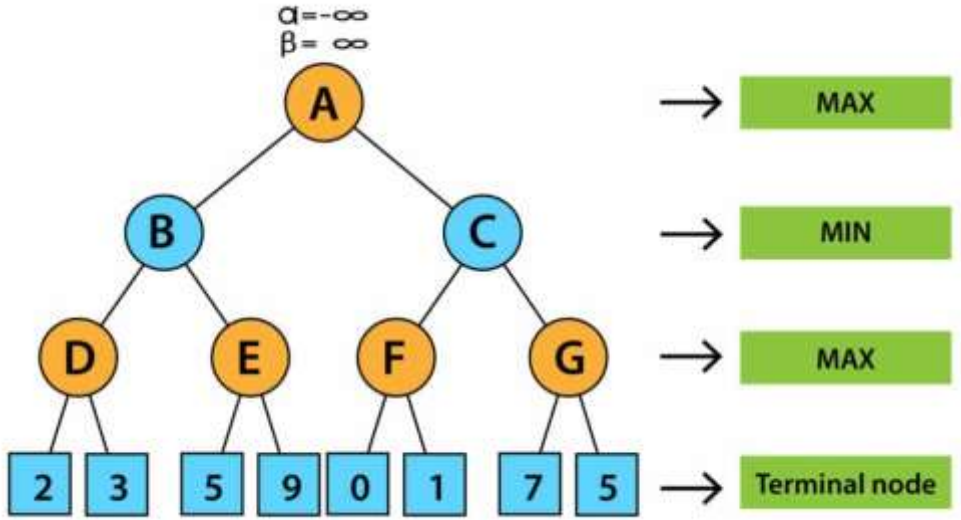
Passing the parameters

- $Alpha(\alpha)$ and $Beta(\beta)$ are passed on to only child nodes.
- While backtracking the game tree, the node values are passed to parent nodes.

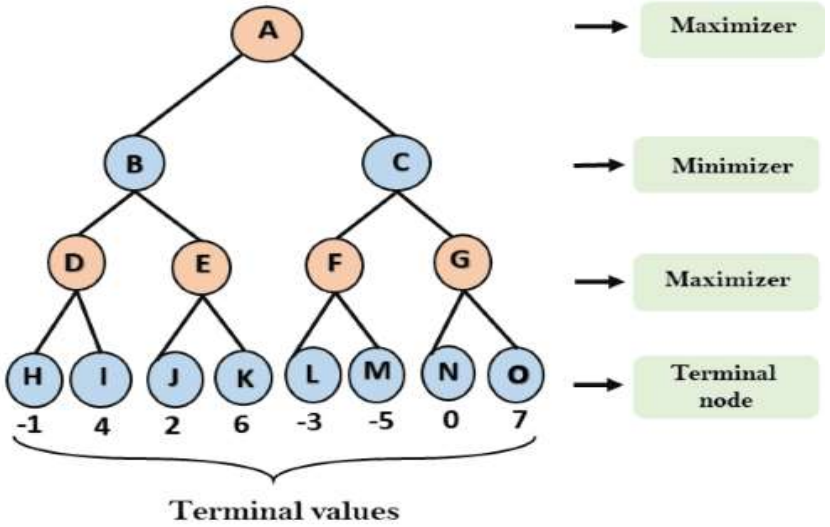
Pruning Condition

- The child sub-trees, which are not yet traversed, are pruned if the condition $\alpha \geq \beta$ holds.

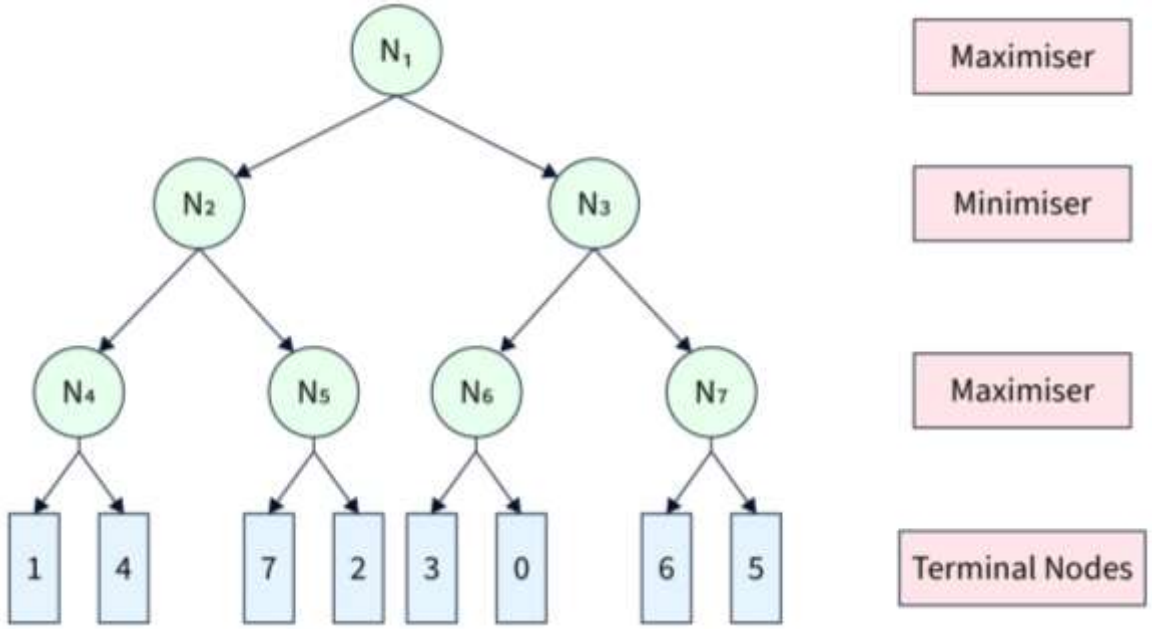
ALPHA BETA PRUNING



ALPHA BETA PRUNING



ALPHA BETA PRUNING



Alpha Beta Pruning